1

Solutions

1.1 **Personal computer (includes workstation and laptop):** Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software.

Personal mobile device (PMD, includes tablets): PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software ("apps") to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input.

Server: Computer used to run large problems and usually accessed via a network.

Warehouse-scale computer: Thousands of processors forming a large cluster.

Supercomputer: Computer composed of hundreds to thousands of processors and terabytes of memory.

Embedded computer: Computer designed to run one application or one set of related applications and integrated into a single system.

1.2

a. Performance via Pipelining
b. Dependability via Redundancy
c. Performance via Prediction
d. Make the Common Case Fast
e. Hierarchy of Memories
f. Performance via Parallelism
g. Design for Moore's Law
h. Use Abstraction to Simplify Design

1.3 The program is compiled into an assembly language program, which is then assembled into a machine language program.

1.4

a. 1280 × 1024 pixels = 1,310,720 pixels => 1,310,720 × 3 = 3,932,160 bytes/frame.
b. 3,932,160 bytes × (8 bits/byte) /100E6 bits/second = 0.31 seconds

1.5

a. performance of P1 (instructions/sec) = $3 \times 10^9/1.5 = 2 \times 10^9$
performance of P2 (instructions/sec) = $2.5 \times 10^9/1.0 = 2.5 \times 10^9$
performance of P3 (instructions/sec) = $4 \times 10^9/2.2 = 1.8 \times 10^9$

b. cycles(P1) $= 10 \times 3 \times 10^9 = 30 \times 10^9$ s

cycles(P2) $= 10 \times 2.5 \times 10^9 = 25 \times 10^9$ s

cycles(P3) $= 10 \times 4 \times 10^9 = 40 \times 10^9$ s

c. No. instructions(P1) $= 30 \times 10^9/1.5 = 20 \times 10^9$

No. instructions(P2) $= 25 \times 10^9/1 = 25 \times 10^9$

No. instructions(P3) $= 40 \times 10^9/2.2 = 18.18 \times 10^9$

$CPI_{new} = CPI_{old} \times 1.2$, then CPI(P1) = 1.8, CPI(P2) = 1.2, CPI(P3) = 2.6

f = No. instr. × CPI/time, then

f(P1) $= 20 \times 10^9 \times 1.8/7 = 5.14$ GHz

f(P2) $= 25 \times 10^9 \times 1.2/7 = 4.28$ GHz

f(P1) $= 18.18 \times 10^9 \times 2.6/7 = 6.75$ GHz

1.6

a. Class A: 10^5 instr. Class B: 2×10^5 instr. Class C: 5×10^5 instr. Class D: 2×10^5 instr.

Time = No. instr. × CPI/clock rate

Total time P1 $= (10^5 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3)/(2.5 \times 10^9) = 10.4 \times 10^{-4}$ s

Total time P2 $= (10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2)/(3 \times 10^9) = 6.66 \times 10^{-4}$ s

CPI(P1) $= 10.4 \times 10^{-4} \times 2.5 \times 10^9/10^6 = 2.6$

CPI(P2) $= 6.66 \times 10^{-4} \times 3 \times 10^9/10^6 = 2.0$

b. clock cycles(P1) $= 10^5 \times 1 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3 = 26 \times 10^5$

clock cycles(P2) $= 10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2 = 20 \times 10^5$

1.7

a. $CPI = T_{exec} \times f/$No. instr.

Compiler A CPI = 1.1

Compiler B CPI = 1.25

b. $f_B/f_A =$ (No. instr.(B) × CPI(B))/(No. instr.(A) × CPI(A)) = 1.37

c. $T_A/T_{new} = 1.67$

$T_B/T_{new} = 2.27$

1.8

1.8.1 $C = 2 \times DP/(V^2 \times F)$

Pentium 4: $C = 3.2E{-}8F$

Core i5 Ivy Bridge: $C = 2.9E{-}8F$

1.8.2 Pentium 4: $10/100 = 10\%$

Core i5 Ivy Bridge: $30/70 = 42.9\%$

1.8.3 $(S_{new} + D_{new})/(S_{old} + D_{old}) = 0.90$

$D_{new} = C \times V_{new}\, 2 \times F$

$S_{old} = V_{old} \times I$

$S_{new} = V_{new} \times I$

Therefore:

$V_{new} = [D_{new}/(C \times F)]1/2$

$D_{new} = 0.90 \times (S_{old} + D_{old}) - S_{new}$

$S_{new} = V_{new} \times (S_{old}/V_{old})$

Pentium 4:

$S_{new} = V_{new} \times (10/1.25) = V_{new} \times 8$

$D_{new} = 0.90 \times 100 - V_{new} \times 8 = 90 - V_{new} \times 8$

$V_{new} = [(90 - V_{new} \times 8)/(3.2E8 \times 3.6E9)]^{1/2}$

$V_{new} = 0.85\,V$

Core i5:

$S_{new} = V_{new} \times (30/0.9) = V_{new} \times 33.3$

$D_{new} = 0.90 \times 70 - V_{new} \times 33.3 = 63 - V_{new} \times 33.3$

$V_{new} = [(63 - V_{new} \times 33.3)/(2.9E8 \times 3.4E9)]^{1/2}$

$V_{new} = 0.64\,V$

1.9

1.9.1

p	# arith inst.	# L/S inst.	# branch inst.	cycles	ex. time	speedup
1	2.56E9	1.28E9	2.56E8	7.94E10	39.7	1
2	1.83E9	9.14E8	2.56E8	5.67E10	28.3	1.4
4	9.12E8	4.57E8	2.56E8	2.83E10	14.2	2.8
8	4.57E8	2.29E8	2.56E8	1.42E10	7.10	5.6

1.9.2

p	ex. time
1	41.0
2	29.3
4	14.6
8	7.33

1.9.3 3

1.10

1.10.1 $\text{die area}_{15\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 7.5^2/84 = 2.10\,\text{cm}^2$

$\text{yield}_{15\text{cm}} = 1/(1 + (0.020 \times 2.10/2))^2 = 0.9593$

$\text{die area}_{20\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 10^2/100 = 3.14\,\text{cm}^2$

$\text{yield}_{20\text{cm}} = 1/(1 + (0.031 \times 3.14/2))^2 = 0.9093$

1.10.2 $\text{cost/die}_{15\text{cm}} = 12/(84 \times 0.9593) = 0.1489$

$\text{cost/die}_{20\text{cm}} = 15/(100 \times 0.9093) = 0.1650$

1.10.3 $\text{die area}_{15\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 7.5^2/(84 \times 1.1) = 1.91\,\text{cm}^2$

$\text{yield}_{15\text{cm}} = 1/(1 + (0.020 \times 1.15 \times 1.91/2))^2 = 0.9575$

$\text{die area}_{20\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 10^2/(100 \times 1.1) = 2.86\,\text{cm}^2$

$\text{yield}_{20\text{cm}} = 1/(1 + (0.03 \times 1.15 \times 2.86/2))^2 = 0.9082$

1.10.4 $\text{defects per area}_{0.92} = (1-y^{.5})/(y^{.5} \times \text{die\_area}/2) = (1 - 0.92^{.5})/(0.92^{.5} \times 2/2) = 0.043\,\text{defects/cm}^2$

$\text{defects per area}_{0.95} = (1-y^{.5})/(y^{.5} \times \text{die\_area}/2) = (1 - 0.95^{.5})/(0.95^{.5} \times 2/2) = 0.026\,\text{defects/cm}^2$

1.11

1.11.1 CPI = clock rate × CPU time/instr. count

clock rate = 1/cycle time = 3 GHz

$\text{CPI(bzip2)} = 3 \times 10^9 \times 750/(2389 \times 10^9) = 0.94$

1.11.2 SPEC ratio = ref. time/execution time

SPEC ratio(bzip2) = 9650/750 = 12.86

1.11.3 CPU time = No. instr. × CPI/clock rate

If CPI and clock rate do not change, the CPU time increase is equal to the increase in the number of instructions, that is 10%.

1.11.4 CPU time(before) = No. instr. × CPI/clock rate

CPU time(after) = 1.1 × No. instr. × 1.05 × CPI/clock rate

CPU time(after)/CPU time(before) = 1.1 × 1.05 = 1.155. Thus, CPU time is increased by 15.5%.

1.11.5 SPECratio = reference time/CPU time

SPECratio(after)/SPECratio(before) = CPU time(before)/CPU time(after) = 1/1.1555 = 0.86. The SPECratio is decreased by 14%.

1.11.6 CPI = (CPU time × clock rate)/No. instr.

CPI = $700 \times 4 \times 10^9/(0.85 \times 2389 \times 10^9) = 1.37$

1.11.7 Clock rate ratio = 4 GHz/3 GHz = 1.33

CPI @ 4 GHz = 1.37, CPI @ 3 GHz = 0.94, ratio = 1.45

They are different because, although the number of instructions has been reduced by 15%, the CPU time has been reduced by a lower percentage.

1.11.8 700/750 = 0.933. CPU time reduction: 6.7%

1.11.9 No. instr. = CPU time × clock rate/CPI

No. instr. = $960 \times 0.9 \times 4 \times 10^9/1.61 = 2146 \times 10^9$

1.11.10 Clock rate = No. instr. × CPI/CPU time.

Clock rate_{new} = No. instr. × CPI/0.9 × CPU time = 1/0.9 clock rate_{old} = 3.33 GHz

1.11.11 Clock rate = No. instr. × CPI/CPU time.

Clock rate_{new} = No. instr. × 0.85 × CPI/0.80 CPU time = 0.85/0.80, clock rate_{old} = 3.18 GHz

1.12

1.12.1 $T(P1) = 5 \times 10^9 \times 0.9/(4 \times 10^9) = 1.125$ s

$T(P2) = 10^9 \times 0.75/(3 \times 10^9) = 0.25$ s

clock rate(P1) > clock rate(P2), performance(P1) < performance(P2)

1.12.2 T(P1) = No. instr. × CPI/clock rate

T(P1) = 2.25 3 1021 s

T(P2) 5 N × 0.75/(3×10^9), then $N = 9 \times 10^8$

1.12.3 MIPS = Clock rate × 10^{-6}/CPI

MIPS(P1) = $4 \times 10^9 \times 10^{-6}/0.9 = 4.44 \times 10^3$

MIPS(P2) = $3 \times 10^9 \times 10^{-6}/0.75 = 4.0 \times 10^3$

MIPS(P1) > MIPS(P2), performance(P1) < performance(P2) (from 11a)

1.12.4 MFLOPS = No. FP operations $\times 10^{-6}$/T

MFLOPS(P1) = .4 × 5E9 × 1E-6/1.125 = 1.78E3

MFLOPS(P2) = .4 × 1E9 × 1E-6/.25 = 1.60E3

MFLOPS(P1) > MFLOPS(P2), performance(P1) < performance(P2) (from 11a)

1.13

1.13.1 $T_{fp} = 70 \times 0.8 = 56$ s. $T_{new} = 56 + 85 + 55 + 40 = 236$ s. Reduction: 5.6%

1.13.2 $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{l/s} + T_{branch} = 165$ s, $T_{int} = 35$ s. Reduction time INT: 58.8%

1.13.3 $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{int} + T_{l/s} = 210$ s. NO

1.14

1.14.1 Clock cycles = CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr.

T_{CPU} = clock cycles/clock rate = clock cycles/2×10^9

clock cycles = 512×10^6; $T_{CPU} = 0.256$ s

To have the number of clock cycles by improving the CPI of FP instructions:

$CPI_{improved\ fp}$ × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr. = clock cycles/2

$CPI_{improved\ fp}$ = (clock cycles/2 − (CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr.)) / No. FP instr.

$CPI_{improved\ fp} = (256 - 462)/50 < 0 = => $ not possible

1.14.2 Using the clock cycle data from a.

To have the number of clock cycles improving the CPI of L/S instructions:

CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{improved\ l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr. = clock cycles/2

$CPI_{improved\ l/s}$ = (clock cycles/2 − (CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + CPI_{branch} × No. branch instr.)) / No. L/S instr.

$CPI_{improved\ l/s} = (256 - 198)/80 = 0.725$

1.14.3 Clock cycles = CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr.

T_{CPU} = clock cycles/clock rate = clock cycles/2×10^9

$CPI_{int} = 0.6 \times 1 = 0.6$; $CPI_{fp} = 0.6 \times 1 = 0.6$; $CPI_{l/s} = 0.7 \times 4 = 2.8$; $CPI_{branch} = 0.7 \times 2 = 1.4$

T_{CPU} (before improv.) = 0.256 s; T_{CPU} (after improv.) = 0.171 s

1.15

processors	exec. time/ processor	time w/overhead	speedup	actual speedup/ideal speedup
1	100			
2	50	54	100/54 = 1.85	1.85/2 = .93
4	25	29	100/29 = 3.44	3.44/4 = 0.86
8	12.5	16.5	100/16.5 = 6.06	6.06/8 = 0.75
16	6.25	10.25	100/10.25 = 9.76	9.76/16 = 0.61

2

Solutions

2.1
```
addi x5, x7,-5
add  x5, x5, x6
[addi f,h,-5 (note, no subi) add f,f,g]
```

2.2
```
f = g+h+i
```

2.3
```
sub     x30, x28, x29    // compute i-j
slli    x30, x30, 3      // multiply by 8 to convert the
                            word offset to a byte offset
ld      x30, 0(x3)       // load A[i-j]
sd      x30, 64(x11)     // store in B[8]
```

2.4
```
B[g]= A[f] + A[f+1]

slli    x30, x5, 3       // x30 = f*8
add     x30, x10, x30    // x30 = &A[f]
slli    x31, x6, 3       // x31 = g*8
add     x31, x11, x31    // x31 = &B[g]
ld      x5, 0(x30)       // f = A[f]
addi    x12, x30, 8      // x12 = &A[f]+8  (i.e. &A[f+1])
ld      x30, 0(x12)      // x30 = A[f+1]
add     x30, x30, x5     // x30 = A[f+1] + A[f]
sd      x30, 0(x31)      // B[g] = x30 (i.e. A[f+1] + A[f])
```

2.5

Little-Endian		Big-Endian	
Address	**Data**	**Address**	**Data**
12	ab	12	12
8	cd	8	ef
4	ef	4	cd
0	12	0	ab

2.6
```
2882400018
```

2.7
```
slli    x28, x28, 3      // x28 = i*8
ld      x28, 0(x10)      // x28 = A[i]
slli    x29, x29, 3      // x29 = j*8
ld      x29, 0(x11)      // x29 = B[j]
add     x29, x28, x29    // Compute x29 = A[i] + B[j]
sd      x29, 64(x11)     // Store result in  B[8]
```

2.8
```
f = 2*(&A)
    addi   x30, x10, 8     // x30 = &A[1]
    addi   x31, x10, 0     // x31 = &A
    sd     x31, 0(x30)     // A[1] = &A
    ld     x30, 0(x30)     // x30 = A[1] = &A
    add    x5, x30, x31    // f = &A + &A = 2*(&A)
```

2.9

	type	opcode, funct3,7	rs1	rs2	rd	imm
`addi x30,x10,8`	I-type	0x13, 0x0, --	10	--	30	8
`addi x31,x10,0`	R-type	0x13, 0x0, --	10	--	31	0
`sd x31,0(x30)`	S-type	0x23, 0x3, --	31	30	--	0
`ld x30,0(x30)`	I-type	0x3, 0x3, --	30	--	30	0
`add x5, x30, x31`	R-type	0x33, 0x0, 0x0	30	31	5	--

2.10

2.10.1 `0x5000000000000000`

2.10.2 `overflow`

2.10.3 `0xB000000000000000`

2.10.4 `no overflow`

2.10.5 `0xD000000000000000`

2.10.6 `overflow`

2.11

2.11.1 There is an overflow if 128 + `x6` > $2^{63} - 1$.
In other words, if `x6` > $2^{63} - 129$.
There is also an overflow if 128 + `x6` < -2^{63}.
In other words, if `x6` < $-2^{63} - 128$ (which is impossible given the range of `x6`).

2.11.2 There is an overflow if 128 – `x6` > $2^{63} - 1$.
In other words, if `x6` < $-2^{63} + 129$.
There is also an overflow if 128 – `x6` < -2^{63}.
In other words, if `x6` > $2^{63} + 128$ (which is impossible given the range of `x6`).

2.11.3 There is an overflow if `x6` – 128 > $2^{63} - 1$.
In other words, if `x6` < $2^{63} + 127$ (which is impossible given the range of `x6`).
There is also an overflow if `x6` – 128 < -2^{63}.
In other words, if `x6` < $-2^{63} + 128$.

2.12 `R-type: add x1, x1, x1`

2.13 S-type: 0x25F3023 (0000 0010 0101 1111 0011 0000 0010 0011)

2.14 R-type: sub x6, x7, x5 (0x40538333: 0100 0000 0101 0011 1000 0011 0011 0011)

2.15 I-type: ld x3, 4(x27) (0x4DB183: 0000 0000 0100 1101 1011 0001 1000 0011)

2.16

2.16.1 The opcode would expand from 7 bits to 9.

The rs1, rs2, and rd fields would increase from 5 bits to 7 bits.

2.16.2 The opcode would expand from 7 bits to 12.

The rs1 and rd fields would increase from 5 bits to 7 bits. This change does not affect the imm field *per se*, but it might force the ISA designer to consider shortening the immediate field to avoid an increase in overall instruction size.

2.16.3 * Increasing the size of each bit field potentially makes each instruction longer, potentially increasing the code size overall.

* However, increasing the number of registers could lead to less register spillage, which would reduce the total number of instructions, possibly reducing the code size overall.

2.17

2.17.1 0x1234567ababefef8

2.17.2 0x2345678123456780

2.17.3 0x545

2.18 It can be done in eight RISC-V instructions:

```
addi x7, x0, 0x3f // Create bit mask for bits 16 to 11
slli x7, x7, 11   // Shift the masked bits
and  x28,x5, x7   // Apply the mask to x5
slli x7, x6, 15   // Shift the mask to cover bits 31
                     to 26
xori x7, x7, -1   // This is a NOT operation
and  x6, x6, x7   // "Zero out" positions 31 to
                     26 of x6
slli x28,x28,15   // Move selection from x5 into
                     positions 31 to 26
or   x6, x6, x28  // Load bits 31 to 26 from x28
```

2.19 xori x5, x6, -1

2.20
```
ld   x6, 0(x17)
slli x6, x6, 4
```

2.21 x6 = 2

2.22

2.22.1 [0x1ff00000, 0x200FFFFE]

2.22.2 [0x1FFFF000, 0x20000ffe]

2.23

2.23.1 The UJ instruction format would be most appropriate because it would allow the maximum number of bits possible for the "loop" parameter, thereby maximizing the utility of the instruction.

2.23.2 It can be done in three instructions:

```
loop:
     addi  x29, x29, -1   // Subtract 1 from x29
     bgt   x29, x0, loop  // Continue if x29 not
                             negative
     addi  x29, x29, 1    // Add back 1 that shouldn't
                             have been subtracted.
```

2.24

2.24.1 The final value of xs is 20.

2.24.2
```
acc = 0;
i = 10;
while (i ! = 0) {
   acc += 2;
   i--;
}
```

2.24.3 4*N + 1 instructions.

2.24.4 (Note: change condition ! = to > = in the while loop)

```
acc = 0;
i = 10;
while (i >= 0) {
   acc += 2;
   i--;
}
```

2.25 The C code can be implemented in RISC-V assembly as follows.

```
LOOPI:
      addi  x7,   x0, 0     // Init i = 0
      bge   x7,   x5, ENDI  // While i < a
      addi  x30,  x10, 0    // x30 = &D
      addi  x29,  x0, 0     // Init j = 0
LOOPJ:
      bge   x29,  x6, ENDJ  // While j < b
      add   x31,  x7, x29   // x31 = i+j
      sd    x31,  0(x30)    // D[4*j] = x31
      addi  x30,  x30, 32   // x30 = &D[4*(j+1)]
      addi  x29,  x29, 1    // j++
      jal   x0,   LOOPJ
ENDJ:
      addi  x7,   x7, 1     // i++;
      jal   x0,   LOOPI
ENDI:
```

2.26 The code requires 13 RISC-V instructions. When a = 10 and b = 1, this results in 123 instructions being executed.

2.27

```
// This C code corresponds most directly to the given
assembly.

int i;
for (i = 0; i < 100; i++) {
  result += *MemArray;
  MemArray++;
}
return result;

// However, many people would write the code this way:

int i;
for (i = 0; i < 100; i++) {
 result += MemArray[i];
}
return result;
```

2.28 The address of the last element of MemArray can be used to terminate the loop:

```
      add x29, x10, 800      // x29 = &MemArray[101]
LOOP:
      ld    x7,   0(x10)
      add   x5,   x5, x7
      addi  x10,  x10, 8
      blt   x10,  x29, LOOP  // Loop until MemArray points
                                to one-past the last element
```

2.29

```
      // IMPORTANT! Stack pointer must reamin a multiple
      of 16!!!!

fib:
      beq    x10,  x0, done  // If n==0, return 0
      addi   x5,   x0, 1
      beq    x10,  x5, done  // If n==1, return 1
      addi   x2,   x2, -16   // Allocate 2 words of stack
                                space
      sd     x1,   0(x2)     // Save the return address
      sd     x10,  8(x2)     // Save the current n
      addi   x10,  x10, -1   // x10 = n-1
      jal    x1,   fib       // fib(n-1)
      ld     x5,   8(x2)     // Load old n from the stack
      sd     x10,  8(x2)     // Push fib(n-1) onto the stack
      addi   x10,  x5, -2    // x10 = n-2
      jal    x1,   fib       // Call fib(n-2)
      ld     x5,   8(x2)     // x5 = fib(n-1)
      add    x10,  x10, x5   // x10 = fib(n-1)+fib(n-2)
// Clean up:
      ld     x1,   0(x2)     // Load saved return address
      addi   x2,   x2, 16    // Pop two words from the stack
done:
      jalr   x0,   x1
```

2.30 [answers will vary]

2.31

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11= c+d from the stack
  jal   x1, g         // Call x10 = g(g(a,b), c+d)
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jalr  x0, x1
```

2.32 We can use the tail-call optimization for the second call to `g`, saving one instruction:

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11 = c+d from the stack
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jal   x0, g         // Call x10 = g(g(a,b), c+d)
```

2.33 *We have no idea what the contents of `x10-x14` are, g can set them as it pleases.

*We don't know what the precise contents of `x8` and `sp` are; but we do know that they are identical to the contents when `f` was called.

*Similarly, we don't know what the precise contents of `x1` are; but, we do know that it is equal to the return address set by the "`jal x1, f`" instruction that invoked `f`.

2.34

```
a_to_i:
  addi   x28,  x0, 10     # Just stores the constant 10
  addi   x29,  x0, 0      # Stores the running total
  addi   x5,   x0, 1      # Tracks whether input is positive
                            or negative
  # Test for initial '+' or '-'
  lbu    x6, 0(x10)       # Load the first character
  addi   x7, x0, 45       # ASCII '-'
  bne    x6, x7, noneg
  addi   x5, x0, -1       # Set that input was negative
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

noneg:
  addi   x7, x0, 43       # ASCII '+'
  bne    x6, x7, main_atoi_loop
  addi   x10, x10, 1      # str++

main_atoi_loop:
  lbu    x6, 0(x10)       # Load the next digit
  beq    x6, x0, done     # Make sure next char is a digit,
                            or fail
  addi   x7, x0, 48       # ASCII '0'
  sub    x6, x6, x7
  blt    x6, x0, fail     # *str < '0'
  bge    x6, x28, fail    # *str >= '9'

  # Next char is a digit, so accumulate it into x29

  mul    x29, x29, x28    # x29 *= 10
  add    x29, x29, x6     # x29 += *str - '0'
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

done:
  addi   x10, x29, 0      # Use x29 as output value
  mul    x10, x10, x5     # Multiply by sign
  jalr   x0, x1           # Return result

fail:
  addi   x10, x0, -1
  jalr   x0, x1
```

2.35

2.35.1 0x11

2.35.2 0x88

2.36
```
lui   x10,  0x11223
addi  x10,  x10, 0x344
slli  x10,  x10, 32
lui   x5,   0x55667
addi  x5,   x5, 0x788
add   x10,  x10, x5
```

2.37
```
setmax:
    try:
    lr.d  x5,  (x10)        # Load-reserve *shvar
    bge   x5,  x11, release # Skip update if *shvar > x
    addi  x5,  x11, 0
release:
    sc.d  x7,  x5, (x10)
    bne   x7,  x0, try      # If store-conditional failed,
                              try again
    jalr  x0,  x1
```

2.38 When two processors A and B begin executing this loop at the same time, at most one of them will execute the store-conditional instruction successfully, while the other will be forced to retry the loop. If processor A's store-conditional successds initially, then B will re-enter the try block, and it will see the new value of shvar written by A when it finally succeeds. The hardware guarantees that both processors will eventually execute the code completely.

2.39

2.39.1 No. The resulting machine would be slower overall.
Current CPU requires (num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 500*1 + 300*10 + 100*3 = 3800 cycles.
The new CPU requires (.75*num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 375*1 + 300*10 + 100*3 = 3675 cycles.
However, given that each of the new CPU's cycles is 10% longer than the original CPU's cycles, the new CPU's 3675 cycles will take as long as 4042.5 cycles on the original CPU.

2.39.2 If we double the performance of arithmetic instructions by reducing their CPI to 0.5, then the the CPU will run the reference program in (500*.5) + (300*10) + 100*3 = 3550 cycles. This represents a speedup of 1.07.

If we improve the performance of arithmetic instructions by a factor of 10 (reducing their CPI to 0.1), then the the CPU will run the reference program in (500*.1) + (300*10) + 100*3 = 3350 cycles. This represents a speedup of 1.13.

2.40

2.40.1 Take the weighted average: 0.7*2 + 0.1*6 + 0.2*3 = 2.6

2.40.2 For a 25% improvement, we must reduce the CPU to 2.6*.75 = 1.95. Thus, we want 0.7*x + 0.1*6 + 0.2*3 < = 1.95. Solving for x shows that the arithmetic instructions must have a CPI of at most 1.07.

2.40.3 For a 50% improvement, we must reduce the CPU to 2.6*.5 = 1.3. Thus, we want 0.7*x + 0.1*6 + 0.2*3 < = 1.3. Solving for x shows that the arithmetic instructions must have a CPI of at most 0.14

2.41

```
ldr   x28, x5(x10), 3      // Load x28=A[f]
addi  x5, x5, 1            // f++
ldr   x29, x5(x10), 3      // Load x29=A[f+1]
add   x29, x29, x28        // Add x29 = A[f] + A[f+1]
sdr   x12, x6(x11), 3      // Store B[g] = x29
```

2.42

```
ldr   x28, x28, (x10), 3   // Load x28=A[i]
ldr   x29, x29, (x11), 3   // Load x29=B[j]
add   x29, x28, x29
sd    x29, 64(x11)         // Store B[8]=x29 (don't
                              need scaled store here)
```

3

Solutions

3.1 5730

3.2 5730

3.3 0101111011010100

The attraction is that each hex digit contains one of 16 different characters (0–9, A–E). Since with 4 binary bits you can represent 16 different patterns, in hex each digit requires exactly 4 binary bits. And bytes are by definition 8 bits long, so two hex digits are all that are required to represent the contents of 1 byte.

3.4 753

3.5 7777 (−3777)

3.6 Neither (63)

3.7 Neither (65)

3.8 Overflow (result = −179, which does not fit into an SM 8-bit format)

3.9 −105 − 42 = −128 (−147)

3.10 −105 + 42 = −63

3.11 151 + 214 = 255 (365)

3.12 62 × 12

Step	Action	Multiplier	Multiplicand	Product
0	Initial Vals	001 010	000 000 110 010	000 000 000 000
1	lsb=0, no op	001 010	000 000 110 010	000 000 000 000
	Lshift Mcand	001 010	000 001 100 100	000 000 000 000
	Rshift Mplier	000 101	000 001 100 100	000 000 000 000
2	Prod=Prod+Mcand	000 101	000 001 100 100	000 001 100 100
	Lshift Mcand	000 101	000 011 001 000	000 001 100 100
	Rshift Mplier	000 010	000 011 001 000	000 001 100 100
3	lsb=0, no op	000 010	000 011 001 000	000 001 100 100
	Lshift Mcand	000 010	000 110 010 000	000 001 100 100
	Rshift Mplier	000 001	000 110 010 000	000 001 100 100
4	Prod=Prod+Mcand	000 001	000 110 010 000	000 111 110 100
	Lshift Mcand	000 001	001 100 100 000	000 111 110 100
	Rshift Mplier	000 000	001 100 100 000	000 111 110 100
5	lsb=0, no op	000 000	001 100 100 000	000 111 110 100
	Lshift Mcand	000 000	011 001 000 000	000 111 110 100
	Rshift Mplier	000 000	011 001 000 000	000 111 110 100
6	lsb=0, no op	000 000	110 010 000 000	000 111 110 100
	Lshift Mcand	000 000	110 010 000 000	000 111 110 100
	Rshift Mplier	000 000	110 010 000 000	000 111 110 100

3.13 62 × 12

Step	Action	Multiplicand	Product/Multiplier
0	Initial Vals	110 010	000 000 001 010
1	lsb=0, no op	110 010	000 000 001 010
	Rshift Product	110 010	000 000 000 101
2	Prod=Prod+Mcand	110 010	110 010 000 101
	Rshift Mplier	110 010	011 001 000 010
3	lsb=0, no op	110 010	011 001 000 010
	Rshift Mplier	110 010	001 100 100 001
4	Prod=Prod+Mcand	110 010	111 110 100 001
	Rshift Mplier	110 010	011 111 010 000
5	lsb=0, no op	110 010	011 111 010 000
	Rshift Mplier	110 010	001 111 101 000
6	lsb=0, no op	110 010	001 111 101 000
	Rshift Mplier	110 010	000 111 110 100

3.14 For hardware, it takes one cycle to do the add, one cycle to do the shift, and one cycle to decide if we are done. So the loop takes (3 × A) cycles, with each cycle being B time units long.

For a software implementation, it takes one cycle to decide what to add, one cycle to do the add, one cycle to do each shift, and one cycle to decide if we are done. So the loop takes (5 × A) cycles, with each cycle being B time units long.

(3 × 8) × 4tu = 96 time units for hardware

(5 × 8) × 4tu = 160 time units for software

3.15 It takes B time units to get through an adder, and there will be A − 1 adders. Word is 8 bits wide, requiring 7 adders. 7 × 4tu = 28 time units.

3.16 It takes B time units to get through an adder, and the adders are arranged in a tree structure. It will require log2(A) levels. An 8 bit wide word requires seven adders in three levels. 3 × 4tu = 12 time units.

3.17 0x33 × 0x55 = 0x10EF. 0x33 = 51, and 51 = 32 + 16 + 2 + 1. We can shift 0x55 left five places (0xAA0), then add 0x55 shifted left four places (0x550), then add 0x55 shifted left once (0xAA), then add 0x55. 0xAA0 + 0x550 + 0xAA + 0x55 = 0x10EF. Three shifts, three adds.

(Could also use 0x55, which is 64 + 16 + 4 + 1, and shift 0x33 left six times, add to it 0x33 shifted left four times, add to that 0x33 shifted left two times, and add to that 0x33. Same number of shifts and adds.)

3.18 74/21 = 3 remainder 9

Step	Action	Quotient	Divisor	Remainder
0	Initial Vals	000 000	010 001 000 000	000 000 111 100
1	Rem=Rem–Div	000 000	010 001 000 000	101 111 111 100
	Rem<0,R+D,Q<<	000 000	010 001 000 000	000 000 111 100
	Rshift Div	000 000	001 000 100 000	000 000 111 100
2	Rem=Rem–Div	000 000	001 000 100 000	111 000 011 100
	Rem<0,R+D,Q<<	000 000	001 000 100 000	000 000 111 100
	Rshift Div	000 000	000 100 010 000	000 000 111 100
3	Rem=Rem–Div	000 000	000 100 010 000	111 100 101 100
	Rem<0,R+D,Q<<	000 000	000 100 010 000	000 000 111 100
	Rshift Div	000 000	000 010 001 000	000 000 111 100
4	Rem=Rem–Div	000 000	000 010 001 000	111 110 110 100
	Rem<0,R+D,Q<<	000 000	000 010 001 000	000 000 111 100
	Rshift Div	000 000	000 001 000 100	000 000 111 100
5	Rem=Rem–Div	000 000	000 001 000 100	111 111 111 000
	Rem<0,R+D,Q<<	000 000	000 001 000 100	000 000 111 100
	Rshift Div	000 000	000 000 100 010	000 000 111 100
6	Rem=Rem–Div	000 000	000 000 100 010	000 000 011 010
	Rem>0,Q<<1	000 001	000 000 100 010	000 000 011 010
	Rshift Div	000 001	000 000 010 001	000 000 011 010
7	Rem=Rem–Div	000 001	000 000 010 001	000 000 001 001
	Rem>0,Q<<1	000 011	000 000 010 001	000 000 001 001
	Rshift Div	000 011	000 000 001 000	000 000 001 001

3.19 In these solutions a 1 or a 0 was added to the Quotient if the remainder was greater than or equal to 0. However, an equally valid solution is to shift in a 1 or 0, but if you do this you must do a compensating right shift of the remainder (only the remainder, not the entire remainder/quotient combination) after the last step.

74/21 = 3 remainder 11

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	010 001	000 000 111 100
1	R<<	010 001	000 001 111 000
	Rem=Rem–Div	010 001	111 000 111 000
	Rem<0,R+D	010 001	000 001 111 000
2	R<<	010 001	000 011 110 000
	Rem=Rem–Div	010 001	110 010 110 000
	Rem<0,R+D	010 001	000 011 110 000
3	R<<	010 001	000 111 100 000
	Rem=Rem–Div	010 001	110 110 110 000
	Rem<0,R+D	010 001	000 111 100 000
4	R<<	010 001	001 111 000 000
	Rem=Rem–Div	010 001	111 110 000 000
	Rem<0,R+D	010 001	001 111 000 000

Step	Action	Divisor	Remainder/Quotient
5	R<<	010 001	011 110 000 000
	Rem=Rem–Div	010 001	111 110 000 000
	Rem>0,R0=1	010 001	001 101 000 001
6	R<<	010 001	011 010 000 010
	Rem=Rem–Div	010 001	001 001 000 010
	Rem>0,R0=1	010 001	001 001 000 011

3.20 201326592 in both cases.

3.21 jal 0x00000000

3.22 0×0C000000 = 0000 1100 0000 0000 0000 0000 0000 0000

= 0 0001 1000 0000 0000 0000 0000 0000 000

sign is positive

exp = 0 × 18 = 24 − 127 = −103

there is a hidden 1

mantissa = 0

answer = 1.0×2^{-103}

3.23 $63.25 \times 10^0 = 111111.01 \times 2^0$

normalize, move binary point five to the left

1.1111101×2^5

sign = positive, exp = 127 + 5 = 132

Final bit pattern: 0 1000 0100 1111 1010 0000 0000 0000 000

= 0100 0010 0111 1101 0000 0000 0000 0000 = 0x427D0000

3.24 $63.25 \times 10^0 = 111111.01 \times 2^0$

normalize, move binary point five to the left

1.1111101×2^5

sign = positive, exp = 1023 + 5 = 1028

Final bit pattern:

0 100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

= 0x404FA00000000000

3.25 $63.25 \times 10^0 = 111111.01 \times 2^0 = 3F.40 \times 16^0$

move hex point two to the left

$0.3F40 \times 16^2$

sign = positive, exp = 64 + 2

Final bit pattern: 01000010001111110100000000000000

3.26 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

move the binary point two to the right

$= -0.101 \times 2^{-2}$

exponent = −2, fraction = −0.10100000000000000000000

answer: 11111111111010110000000000000000000

3.27 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

move the binary point three to the right, $= -1.01 \times 2^{-3}$

exponent = −3 = −3 + 15 = 12, fraction = −0.0100000000

answer: 1011000100000000

3.28 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

move the binary point two to the right

$= -0.101 \times 2^{-2}$

exponent = −2, fraction = −0.10100000000000000000000000

answer: 10110000000000000000000000000101

3.29 $2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$

$2.6125 \times 10^1 = 26.125 = 11010.001 = 1.1010001000 \times 2^4$

$4.150390625 \times 10^{-1} = 0.4150390625 = 0.011010100111 = 1.1010100111 \times 2^{-2}$

Shift binary point six to the left to align exponents,

```
             GR
1.1010001000 00
1.0000011010 10 0111 (Guard 5 1, Round 5 0,
Sticky 5 1)
-------------------
1.1010100010 10
```

In this case the extra bit (G,R,S) is more than half of the least significant bit (0).

Thus, the value is rounded up.

$1.1010100011 \times 2^4 = 11010.100011 \times 2^0 = 26.546875 = 2.6546875 \times 10^1$

3.30 $-8.0546875 \times -1.79931640625 \times 10^{-1}$

$-8.0546875 = -1.0000000111 \times 2^3$

$-1.79931640625 \times 10^{-1} = -1.0111000010 \times 2^{-3}$

Exp: $-3 + 3 = 0$, $0 + 16 = 16$ (10000)

Signs: both negative, result positive

Fraction:

```
            1.0000000111
          × 1.0111000010
            ------------
             00000000000
            10000000111
           00000000000
          00000000000
         00000000000
        00000000000
       10000000111
      10000000111
     10000000111
    00000000000
   10000000111
1.01110011000001001110
```

1.0111001100 00 01001110 Guard = 0, Round = 0, Sticky = 1:NoRnd

$1.0111001100 \times 2^0 = 0100000111001100$ ($1.0111001100 = 1.44921875$)

$-8.0546875 \times -0.179931640625 = 1.4492931365966796875$

Some information was lost because the result did not fit into the available 10-bit field. Answer (only) off by 0.0000743865966796875

3.31 $8.625 \times 10^1 / -4.875 \times 10^0$

$8.625 \times 10^1 = 1.0101100100 \times 2^6$

$-4.875 = -1.0011100000 \times 2^2$

Exponent = 6 − 2 = 4, 4 + 15 = 19 (10011)

Signs: one positive, one negative, result negative

Fraction:

```
                          1.00011011000100111
10011100000. | 10101100100.0000000000000000
             −10011100000.
              --------------
                10000100.0000
               −1001110.0000
               --------------
                 1100110.00000
                −100111.00000
                --------------
                    1111.0000000
                  −1001.1100000
                  -------------
                     101.01000000
                   −100.11100000
                   -------------
                     000.011000000000
                       −.010011100000
                        --------------
                         .000100100000000
                       −.000010011100000
                        -----------------
                         .0000100001000000
                       −.0000010011100000
                        ------------------
                         .00000011011000000
                       −.00000010011100000
                        --------------------
                         .00000000110000000
```

1.000110110001001111 Guard = 0, Round = 1, Sticky = 1: No Round, fix sign

$-1.0001101100 \times 2^4 = 1101000001101100 = 10001.101100 = -17.6875$

$86.25/-4.875 = -17.692307692307$

Some information was lost because the result did not fit into the available 10-bit field. Answer off by 0.00480769230

3.32 $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3)$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^3 = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(A)        1.1001100000
(B)       +1.0110000000
           -------------
           10.1111100000 Normalize,
(A+B)       1.0111110000 × 2^-1
(C)       +1.1011101011
(A+B)       .0000000000 10 111110000 Guard = 1,
                        Round = 0, Sticky = 1
          ---------------
(A+B)+C  +1.1011101011 10 1 Round up
```

$(A+B)+C = 1.1011101100 \times 2^{10} = 0110101011101100 = 1772$

3.33 $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^3 = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(B)         .0000000000 01 0110000000 Guard = 0,
                           Round = 1, Sticky = 1
(C)       +1.1011101011
           -------------
(B+C)     +1.1011101011
```

```
(A)       .0000000000  011001100000
       --------------
A + (B + C) + 1.1011101011 No round
```

A + (B + C) + $1.1011101011 \times 2^{10}$ = 0110101011101011 = 1771

3.34 No, they are not equal: (A+B)+C = 1772, A+(B+C) = 1771 (steps shown above).

Exact: 0.398437 + 0.34375 + 1771 = 1771.742187

3.35 $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2}$

(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$

(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$

(C) $1.05625 \times 10^{2} = 1.1010011010 \times 2^{6}$

Exp: −9−8 = −17

Signs: both positive, result positive

Fraction:

```
(A)                  1.1100000000
(B)                × 1.0001000000
                     --------------
                 11100000000
              11100000000
             ----------------------
          1.11011100000000000000
A×B       1.1101110000 00 00000000
Guard = 0, Round = 0, Sticky = 0: No Round
```

A×B $1.1101110000 \times 2^{-17}$ UNDERFLOW: Cannot represent number

3.36 $3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$

(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$

(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$

(C) $1.05625 \times 10^{2} = 1.1010011010 \times 2^{6}$

Exp: − 8 + 6 = −2

Signs: both positive, result positive

Fraction:

```
(B)    1.0001000000
(C)  × 1.1010011010
          -------------
           10001000000
         10001000000
        10001000000
      10001000000
   10001000000
  10001000000
 -----------------------
1.11000000111010000000 0
1.1100000011 10 100000000 Guard 5 1, Round 5 0, Sticky
5 1: Round
```

$B \times C \quad 1.1100000100 \times 2^{-2}$

Exp: $-9-2 = -11$

Signs: both positive, result positive

Fraction:

```
(A)     1.1100000000
(B × C)× 1.1100000100
          -------------
           11100000000
     11100000000
    11100000000
   11100000000
  ---------------------
11.000100011100000000 0    Normalize, add 1 to exponent
 1.1000100011 10 0000000000 Guard=1, Round=0, Sticky=0:
Round to even
```

$A \times (B \times C)\ 1.1000100100 \times 2^{-10}$

3.37 b) No:

$A \times B = 1.1101110000 \times 2^{-17}$ UNDERFLOW: Cannot represent

$A \times (B \times C) = 1.1000100100 \times 2^{-10}$

A and B are both small, so their product does not fit into the 16-bit floating point format being used.

3.38 $1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$

(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$

(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exponents match, no shifting necessary

```
(B)      1.0011010011
(C)     −1.0011010010
        ----------------
(B + C)  0.0000000001 × 2^14
(B + C)  1.0000000000 × 2^4
```

Exp: 0 + 4 = 4

Signs: both positive, result positive

Fraction:

```
(A)                    1.1010101010
(B + C)              × 1.0000000000
                       ------------
           11010101010
          -----------------------
          1.10101010100000000000
A × (B + C) 1.1010101010 0000000000 Guard = 0, Round =
          0, sticky = 0: No round
```

$A \times (B + C)$ 1.1010101010×2^4

3.39 $1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$

(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$

(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exp: 0 + 14 = 14

Signs: both positive, result positive

Fraction:

```
(A)              1.1010101010
(B)            × 1.0011010011
                 ------------
                  11010101010
                 11010101010
                11010101010
               11010101010
              11010101010
             11010101010
            ------------------------------
            10.0000001001100001111 Normalize, add 1 to
                                   exponent
A × B    1.0000000100 11 00001111 Guard = 1, Round = 1,
                                    Sticky = 1: Round
A × B    1.0000000101 × 2^15
```

Exp: 0 + 14 = 14

Signs: one negative, one positive, result negative

Fraction:

```
(A)                              1.1010101010
(C)                            × 1.0011010010
                                 ------------
                                 11010101010
                                11010101010
                              11010101010
                             11010101010
                           11010101010
                          -------------------------
                          10.0000000111110111010
Normalize, add 1 to exponent
A × C                     1.0000000011 11 101110100
Guard = 1, Round = 1, Sticky = 1: Round
A × C       −1.0000000100 × 2^15
A × B               1.0000000101 × 2^15
A × C              −1.0000000100 × 2^15
                     --------------
A × B+A × C           .0000000001 × 2^15
A × B+A × C          1.0000000000 × 2^5
```

3.40 b) No:

$A\times(B+C) = 1.1010101010 \times 2^4 = 26.65625$, and $(A\times B)+(A\times C) = 1.0000000000 \times 2^5 = 32$

Exact: $1.666015625 \times (19{,}760 - 19{,}744) = 26.65625$

3.41

Answer	sign	exp	Exact?
1 01111101 00000000000000000000000	–	−2	Yes

3.42 b+b+b+b = −1

b × 4 = −1

They are the same

3.43 0101 0101 0101 0101 0101 0101

No

3.44 0011 0011 0011 0011 0011 0011

No

3.45 0101 0000 0000 0000 0000 0000

0.5

Yes

3.46 01010 00000 00000 00000

0.A

Yes

3.47 Instruction assumptions:

(1) 8-lane 16-bit multiplies

(2) sum reductions of the four most significant 16-bit values

(3) shift and bitwise operations

(4) 128-, 64-, and 32-bit loads and stores of most significant bits

Outline of solution:

```
load register F[bits 127:0] = f[3..0] & f[3..0] (64-bit
load)
load register A[bits 127:0] = sig_in[7..0] (128-bit load)
```

```
 for i = 0 to 15 do
      load register B[bits 127:0] = sig_in[(i*8+7..i*8]
      (128-bit load)

      for j = 0 to7 do
      (1) eight-lane multiply C[bits 127:0] = A*F
      (eight 16-bit multiplies)
      (2) set D[bits 15:0] = sum of the four 16-bit values
      in C[bits 63:0] (reduction of four 16-bit values)
         (3) set D[bits 31:16] = sum of the four 16-bit
         values in C[bits 127:64] (reduction of four 16-
         bit values)
         (4) store D[bits 31:0] to sig_out (32-bit store)
         (5) set A = A shifted 16 bits to the left
         (6) set E = B shifted 112 shifts to the right
         (7) set A = A OR E
         (8) set B = B shifted 16 bits to the left
 end for
end for
```

4

Solutions

4.1

4.1.1 The value of the signals is as follows:

RegWrite	ALUSrc	ALUoperation	MemWrite	MemRead	MemToReg
true	0	"and"	false	false	0

Mathematically, the `MemRead` control wire is a "don't care": the instruction will run correctly regardless of the chosen value. Practically, however, `MemRead` should be set to false to prevent causing a segment fault or cache miss.

4.1.2 Registers, ALUsrc mux, ALU, and the MemToReg mux.

4.1.3 All blocks produce some output. The outputs of DataMemory and Imm Gen are not used.

4.2 `Reg2Loc` for `ld`: When executing `ld`, it doesn't matter which value is passed to "Read register 2", because the ALUSrc mux ignores the resulting "Read data 2" output and selects the sign extended immediate value instead.

`MemToReg` for `sd` and `beq`: Neither `sd` nor `beq` write a value to the register file. It doesn't matter which value the MemToReg mux passes to the register file because the register file ignores that value.

4.3

4.3.1 25 + 10 = 35%. Only `Load` and `Store` use Data memory.

4.3.2 100% Every instruction must be fetched from instruction memory before it can be executed.

4.3.3 28 + 25 + 10 + 11 + 2 = 76%. Only R-type instructions do not use the Sign extender.

4.3.4 The sign extend produces an output during every cycle. If its output is not needed, it is simply ignored.

4.4

4.4.1 Only `loads` are broken. `MemToReg` is either 1 or "don't care" for all other instructions.

4.4.2 I-type, `loads`, `stores` are all broken.

4.5 For context: The encoded instruction is `sd x12, 20(x13)`

4.5.1

ALUop	ALU Control Lines
00	0010

4.5.2 The new PC is the old PC + 4. This signal goes from the PC, through the "PC + 4" adder, through the "branch" mux, and back to the PC.

4.5.3 `ALUsrc`: Inputs: `Reg[x12]` and `0x0000000000000014`; Output: `0x0000000000000014`

`MemToReg`: Inputs: `Reg[x13] + 0x14` and `<undefined>`; output: `<undefined>`

`Branch`: Inputs: `PC+4` and `0x000000000000000A`

4.5.4 ALU inputs: `Reg[x13]` and `0x0000000000000014`

PC + 4 adder inputs: `PC` and `4`

Branch adder inputs: `PC` and `0x0000000000000028`

4.6

4.6.1 No additional logic blocks are needed.

4.6.2
`Branch: false`
`MemRead: false` (See footnote from solution to problem 4.1.1.)
`MemToReg: 0`
`ALUop: 10` (or simply saying "add" is sufficient for this problem)
`MemWrite: false`
`ALUsrc: 1`
`RegWrite: 1`

4.7

4.7.1 `R-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.2 `ld`: 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950 ps

4.7.3 `sd`: 30 + 250 + 150 + 200 + 25 + 250 = 905

4.7.4 `beq`: 30 + 250 + 150 + 25 + 200 + 5 + 25 + 20 = 705

4.7.5 `I-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.6 950ps

4.8 Using the results from Problem 4.7, we see that the average time per instruction is

.52*700 + .25*950 + .11*905 + .12 * 705 = 785.6ps

In contrast, a single-cycle CPU with a "normal" clock would require a clock cycle time of 950.

Thus, the speedup would be 925/787.6 = 1.174

4.9

4.9.1 Without improvement: 950; With improvement: 1250

4.9.2 The running time of a program on the original CPU is 950*n. The running time on the improved CPU is 1250*(0.95)*n = 1187.5. Thus, the "speedup" is 0.8. (Thus, this "improved" CPU is actually slower than the original).

4.9.3 Because adding a multiply instruction will remove 5% of the instructions, the cycle time can grow to as much as 950/(0.95) = 1000. Thus, the time for the ALU can increase by up to 50 (from 200 to 250).

4.10

4.10.1 The additional registers will allow us to remove 12% of the loads and stores, or (0.12)*(0.25 + 0.1) = 4.2% of all instructions. Thus, the time to run n instructions will decrease from 950*n to 960*.958*n = 919.68*n. That corresponds to a speedup of 950/895.73 = 1.03.

4.10.2 The cost of the original CPU is 4507; the cost of the improved CPU is 4707.

PC: 5
I-Mem: 1000
Register file: 200
ALU: 100
D-Mem: 2000
Sign Extend: 1002
Controls: 10002
adders: 30*24
muxes: 4*102
single gates: 2*1

Thus, for a 3% increase in performance, the cost of the CPU increases by about 4.4%.

4.10.3 From a strictly mathematical standpoint it does not make sense to add more registers because the new CPU costs more per unit of performance. However, that simple calculation does not account for the utility of the performance. For example, in a real-time system, a 3% performance may make the difference between meeting or missing deadlines. In which case, the improvement would be well worth the 4.4% additional cost.

4.11

4.11.1 No new functional blocks are needed.

4.11.2 Only the control unit needs modification.

4.11.3 No new data paths are needed.

4.11.4 No new signals are needed.

4.12

4.12.1 No new functional blocks are needed.

4.12.2 The register file needs to be modified so that it can write to two registers in the same cycle. The ALU would also need to be modified to allow read data 1 or 2 to be passed through to write data 1.

4.12.3 The answer depends on the answer given in 4.12.2: whichever input was not allowed to pass through the ALU above must now have a data path to write data 2.

4.12.4 There would need to be a second RegWrite control wire.

4.12.5 Many possible solutions.

4.13

4.13.1 We need some additional muxes to drive the data paths discussed in 4.13.3.

4.13.2 No functional blocks need to be modified.

4.13.3 There needs to be a path from the ALU output to data memory's write data port. There also needs to be a path from read data 2 directly to Data memory's Address input.

4.13.4 These new data paths will need to be driven by muxes. These muxes will require control wires for the selector.

4.13.5 Many possible solutions.

4.14 None: all instructions that use sign extend also use the register file, which is slower.

4.15

4.15.1 The new clock cycle time would be 750. ALU and Data Memory will now run in parallel, so we have effectively removed the faster of the two (the ALU with time 200) from the critical path.

4.15.2 Slower. The original CPU takes 950*n picoseconds to run n instructions. The same program will have approximately 1.35*n instructions when compiled for the new machine. Thus, the time on the new machine will be 750*1.35n = 1012.5*n. This represents a "speedup" of .93.

4.15.3 The number of loads and stores is the primary factor. How the loads and stores are used can also have an effect. For example, a program whose loads and stores tend to be to only a few different address may also run faster on the new machine.

4.15.4 This answer is a matter of opinion.

4.16

4.16.1 Pipelined: 350; non-pipelined: 1250

4.16.2 Pipelined: 1250; non-pipelined: 1250

4.16.3 Split the ID stage. This reduces the clock-cycle time to 300ps.

4.16.4 35%.

4.16.5 65%

4.17 $n + k - 1$. Let's look at when each instruction is in the WB stage. In a k-stage pipeline, the 1st instruction doesn't enter the WB stage until cycle k. From that point on, at most one of the remaining $n - 1$ instructions is in the WB stage during every cycle.

This gives us a minimum of $k + (n - 1) = n + k - 1$ cycles.

4.18 x13 = 33 and x14 = 36

4.19 x15 = 54 (The code will run correctly because the result of the first instruction is written back to the register file at the beginning of the 5th cycle, whereas the final instruction reads the updated value of x1 during the second half of this cycle.)

4.20
```
addi x11, x12, 5
NOP
NOP
add x13, x11, x12
addi x14, x11, 15
NOP
add x15, x13, x12
```

4.21

4.21.1 Pipeline without forwarding requires 1.4*n*250ps. Pipeline with forwarding requires 1.05*n*300ps. The speedup is therefore (1.4*250)/(1.05*300) = 1.11.

4.21.2 Our goal is for the pipeline with forwarding to be faster than the pipeline without forwarding. Let y be the number of stalls remaining as a percentage of "code" instructions. Our goal is for 300*(1+y)*n < 250*1.4*n. Thus, y must be less than 16.7%.

4.21.3 This time, our goal is for 300(1 + y)*n < 250(1 + x)*n. This happens when y < (250x − 50)/300.

4.21.4 It cannot. In the best case, where forwarding eliminates the need for every NOP, the program will take time 300*n to run on the pipeline with forwarding. This is slower than the 250*1.075*n required on the pipeline with no forwarding.

4.21.5 Speedup is not possible when the solution to 4.21.3 is less than 0. Solving 0< (250x − 50)/300 for x gives that x must be at least 0.2.

4.22

4.22.1 Stalls are marked with **:

```
sd  x29, 12(x16)  IF ID EX ME WB
ld  x29, 8(x16)      IF ID EX ME WB
sub x17, x15, x14       IF ID EX ME WB
bez x17, label             ** ** IF ID EX ME WB
add x15, x11, x14                    IF ID EX ME WB
sub x15,x30,x14                         IF ID EX ME WB
```

4.22.2 Reordering code won't help. Every instruction must be fetched; thus, every data access causes a stall. Reordering code will just change the pair of instructions that are in conflict.

4.22.3 You can't solve this structural hazard with NOPs, because even the NOPs must be fetched from instruction memory.

4.22.4 35%. Every data access will cause a stall.

4.23

4.23.1 The clock period won't change because we aren't making any changes to the slowest stage.

4.23.2 Moving the MEM stage in parallel with the EX stage will eliminate the need for a cycle between loads and operations that use the result of the loads. This can potentially reduce the number of stalls in a program.

4.23.3 Removing the offset from `ld` and `sd` may increase the total number of instructions because some `ld` and `sd` instructions will need to be replaced with a `addi/ld` or `addi/sd` pair.

4.24 The second one. A careful examination of Figure 4.59 shows that the need for a stall is detected during the ID stage. It is this stage that prevents the fetch of a new instruction, effectively causing the add to repeat its ID stage.

4.25

4.25.1 … indicates a stall. ! indicates a stage that does not do useful work.

```
ld x10, 0(x13)     IF ID EX ME | WB
ld x11, 8(x13)        IF ID EX | ME WB
add x12, x10, x11        IF ID | .. EX ME! WB
addi x13, x13, -16          IF | .. ID EX  ME! WB
bnez x12, LOOP                 | .. IF ID  EX  ME! WB!
ld x10, 0(x13)                       IF  ID  EX  ME  WB
ld x11, 8(x13)                           IF  ID  EX  ME WB
add x12, x10, x11                            IF  ID  .. EX | ME! WB
addi x13, x13, -16                               IF  .. ID | EX  ME! WB
bnez x12, LOOP                                       IF | ID  EX  ME! WB!
Completely busy               | N  N  N   N   N   N   N  N  |
```

4.25.2 In a particular clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. As the diagram above shows, there are not any cycles during which every pipeline stage is doing useful work.

4.26

4.26.1
```
// EX to 1st only:
add x11, x12, x13
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only:
ld x11, 0(x12)
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only:
add x11, x12, x13
add x5, x6, x7
add x14, x11, x12

// MEM to 2nd only:
ld x11, 0(x12)
add x5, x6, x7
add x14, x11, x13

// EX to 1st and EX to 2nd:
add x11, x12, x13
add x5, x11, x15
add x16, x11, x12
```

4.26.2
```
// EX to 1st only: 2 nops
add x11, x12, x13
nop
nop
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only: 2 stalls
ld x11, 0(x12)
nop
nop
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only: 1 nop
add x11, x12, x13
add x5, x6, x7
nop
add x14, x11, x12
```

```
// MEM to 2nd only: 1 nop
ld x11, 0(x12)
add x5, x6, x7
nop
add x14, x11, x13

// EX to 1st and EX to 2nd: 2 nops
add x11, x12, x13
nop
nop
add x5, x11, x15
add x16, x11, x12
```

4.26.3 Consider this code:

```
ld  x11, 0(x5)       # MEM to 2nd --- one stall
add x12, x6, x7      # EX to 1st  --- two stalls
add x13, x11, x12
add x28, x29, x30
```

If we analyze each instruction separately, we would calculate that we need to add 3 stalls (one for a "MEM to 2nd" and two for an "EX to 1st only". However, as we can see below, we need only two stalls:

```
ld x11, 0(x5)
add  x12, x6, x7
nop
nop
add  x13, x11, x12
add  x28, x29, x30
```

4.26.4 Taking a weighted average of the answers from 4.26.2 gives 0.05*2 + 0.2*2 + 0.05*1 + 0.1*1 + 0.1*2 = 0.85 stalls per instruction (on average) for a CPI of 1.85. This means that 0.85/1.85 cycles, or 46%, are stalls.

4.26.5 The only dependency that cannot be handled by forwarding is from the MEM stage to the next instruction. Thus, 20% of instructions will generate one stall for a CPI of 1.2. This means that 0.2 out of 1.2 cycles, or 17%, are stalls.

4.26.6 If we forward from the EX/MEM register only, we have the following stalls/NOPs

EX to 1st:	0
MEM to 1st:	2
EX to 2nd:	1
MEM to 2nd:	1
EX to 1st and 2nd:	1

This represents an average of 0.05*0 + 0.2*2 + 0.05*1 + 0.10*1 + 0.10*1 = 0.65 stalls/instruction. Thus, the CPI is 1.65

IF we forward from MEM/WB only, we have the following stalls/NOPs

EX to 1st:	1
MEM to 1st:	1
EX to 2nd:	0
MEM to 2nd:	0
EX to 1st and 2nd:	1

This represents an average of 0.05*1 + 0.2*1 + 0.1*1 = 0.35 stalls/instruction. Thus, the CPI is 1.35.

4.26.7

	No forwarding	EX/MEM	MEM/WB	Full Forwarding
CPI	1.85	1.65	1.35	1.2
Period	120	120	1.20	130
Time	222n	198n	162n	156n
Speedup	–	1.12	1.37	1.42

4.26.8 CPI for full forwarding is 1.2
CPI for "time travel" forwarding is 1.0
clock period for full forwarding is 130
clock period for "time travel" forwarding is 230

Speedup = (1.2*130)/ (1*230) = 0.68 (That means that "time travel" forwarding actually slows the CPU.)

4.26.9 When considering the "EX/MEM" forwarding in 4.26.6, the "EX to 1st" generates no stalls, but "EX to 1st and EX to 2nd" generates one stall. However, "MEM to 1st" and "MEM to 1st and MEM to 2nd" will always generate the same number of stalls. (All "MEM to 1st" dependencies cause a stall, regardless of the type of forwarding. This stall causes the 2nd instruction's ID phase to overlap with the base instruction's WB phase, in which case no forwarding is needed.)

4.27

4.27.1
```
add   x15, x12, x11
nop
nop
ld    x13, 4(x15)
ld    x12, 0(x2)
nop
or    x13, x15, x13
nop
nop
sd    x13, 0(x15)
```

4.27.2 It is not possible to reduce the number of NOPs.

4.27.3 The code executes correctly. We need hazard detection only to insert a stall when the instruction following a load uses the result of the load. That does not happen in this case.

4.27.4

Cycle	1	2	3	4	5	6	7	8	
add	IF	ID	EX	ME	WB				
ld		IF	ID	EX	ME	WB			
ld			IF	ID	EX	ME	WB		
or				IF	ID	EX	ME	WB	
sd					IF	ID	EX	ME	WB

Because there are no stalls in this code, PCWrite and IF/IDWrite are always 1 and the mux before ID/EX is always set to pass the control values through.

(1) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(2) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(3) ForwardA = 0; ForwardB = 0 (no forwarding; values taken from registers)

(4) ForwardA = 2; ForwardB = 0 (base register taken from result of previous instruction)

(5) ForwardA = 1; ForwardB = 1 (base reguster taken from result of two instructions previous)

(6) ForwardA = 0; ForwardB = 2 (rs1 = x15 taken from register; rs2 = x13 taken from result of 1st ld—two instructions ago)

(7) ForwardA = 0; ForwardB = 2 (base register taken from register file. Data to be written taken from previous instruction)

4.27.5 The hazard detection unit additionally needs the values of rd that comes out of the MEM/WB register. The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by (or forwarded from) the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. The Hazard unit already has the value of `rd` from the EX/MEM register as inputs, so we need only add the value from the MEM/WB register.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

The value of `rd` from EX/MEM is needed to detect the data hazard between the `add` and the following `ld`. The value of `rd` form MEM/WB is needed to detect the data hazard between the first `ld` instruction and the `or` instruction.

4.27.6

Cycle	1	2	3	4	5	6
add	IF	ID	EX	ME	WB	
ld		IF	ID	–	–	EX
ld			IF	–	–	ID

(1) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(2) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(3) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(4) PCWrite = 0; IF/IDWrite = 0; control mux = 1

(5) PCWrite = 0; IF/IDWrite = 0; control mux = 1

4.28

4.28.1 The CPI increases from 1 to 1.4125.

An incorrectly predicted branch will cause three instructions to be flushed: the instructions currently in the IF, ID, and EX stages. (At this point, the branch instruction reaches the MEM stage and updates the PC with the correct next instruction.) In other words, 55% of the branches will result in the flushing of three instructions, giving us a CPI of $1 + (1 - 0.45)(0.25)3 = 1.4125$. (Just to be clear: the always-taken predictor is correct 45% of the time, which means, of course, that it is incorrect $1 - 0.45 = 55\%$ of the time.)

4.28.2 The CPI increases from 1 to 1.3375. $(1 + (.25)(1 - .55) = 1.1125)$

4.28.3 The CPI increases from 1 to 1.1125. $(1 + (.25)(1 - .85) = 1.0375)$

4.28.4 The speedup is approximately 1.019.

Changing half of the branch instructions to an ALU instruction reduces the percentage of instructions that are branches from 25% to 12.5%. Because predicted and mispredicted branches are replaced equally, the misprediction rate remains 15%. Thus, the new CPU is $1 + (.125)(1 - .85) = 1.01875$. This represents a speedup of 1.0375 / 1.01875 = 1.0184

4.28.5 The "speedup" is .91.

There are two ways to look at this problem. One way is to look at the two ADD instruction as a branch with an "extra" cycle. Thus, half of the branches have 1 extra cycle; 15% of the other half have 1 extra cycles (the pipeline flush); and the remaining branches (those correctly predicted) have no extra cycles. This gives us a CPI of $1 + (.5)(.25)*1 + (.5)(.25)(.15)*1 = 1.14375$ and a speedup of 1.0375 / 1.14375 = .91.

We can also treat the ADD instructions as separate instructions. The modified program now has 1.125n instructions (half of 25% produce

an extra instruction). .125n of these 1.125n instruction (or 11.1%) are branches. The CPI for this new program is 1 + (.111)(.15)*1 = 1.01665. When we factor in the 12.5% increase in instructions, we get a speedup of 1.0375 / (1.125 * 1.01665) = .91.

4.28.6 The predictor is 25% accurate on the remaining branches. We know that 80% of branches are always predicted correctly and the overall accuracy is 0.85. Thus, 0.8*1 + 0.2*x = 0.85. Solving for x shows that x = 0.25.

4.29

4.29.1

Always Taken	Always not-taken
3/5 = 60%	2/5 = 40%

4.29.2

Outcomes	Predictor value at time of prediction	Correct of Incorrect	Accuracy
T, NT, T, T	0,1,0,1	I,C,I,I	25%

4.29.3 The first few recurrences of this pattern do not have the same accuracy as the later ones because the predictor is still warming up. To determine the accuracy in the "steady state", we must work through the branch predictions until the predictor values start repeating (i.e. until the predictor has the same value at the start of the current and the next recurrence of the pattern).

Outcomes	Predictor value at time of prediction	Correct of Incorrect (in steady state)	Accuracy
T, NT, T, T, NT	1st occurrence: 0,1,0,1,2 2nd occurrence: 1,2,1,2,3 3rd occurrence: 2,3,2,3,3 4th occurrence: 2,3,2,3,3	C,I,C,C,I	60%

4.29.4 The predictor should be an N-bit shift register, where N is the number of branch outcomes in the target pattern. The shift register should be initialized with the pattern itself (0 for NT, 1 for T), and the prediction is always the value in the leftmost bit of the shift register. The register should be shifted after each predicted branch.

4.29.5 Since the predictor's output is always the opposite of the actual outcome of the branch instruction, the accuracy is zero.

4.29.6 The predictor is the same as in part d, except that it should compare its prediction to the actual outcome and invert (logical NOT) all the bits in the shift register if the prediction is incorrect. This predictor still always perfectly predicts the given pattern. For the opposite pattern, the first prediction will be incorrect, so the predictor's state is inverted and after

that the predictions are always correct. Overall, there is no warm-up period for the given pattern, and the warm-up period for the opposite pattern is only one branch.

4.30

4.30.1

Instruction 1	Instruction 2
Invalid target address (EX)	Invalid data address (MEM)

4.30.2 The Mux that selects the next PC must have inputs added to it. Each input is a constant address of an exception handler. The exception detectors must be added to the appropriate pipeline stage and the outputs of these detectors must be used to control the pre-PC Mux, and also to convert to NOPs instructions that are already in the pipeline behind the exception-triggering instruction.

4.30.3 Instructions are fetched normally until the exception is detected. When the exception is detected, all instructions that are in the pipeline after the first instruction must be converted to NOPs. As a result, the second instruction never completes and does not affect pipeline state. In the cycle that immediately follows the cycle in which the exception is detected, the processor will fetch the first instruction of the exception handler.

4.30.4 This approach requires us to fetch the address of the handler from memory. We must add the code of the exception to the address of the exception vector table, read the handler's address from memory, and jump to that address. One way of doing this is to handle it like a special instruction that puts the address in EX, loads the handler's address in MEM, and sets the PC in WB.

4.30.5 We need a special instruction that allows us to move a value from the (exception) Cause register to a general-purpose register. We must first save the general-purpose register (so we can restore it later), load the Cause register into it, add the address of the vector table to it, use the result as an address for a load that gets the address of the right exception handler from memory, and finally jump to that handler.

4.31.1

```
                    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
li x12, 0           IF ID EX ME WB
jal ENT             IF ID .. EX ME WB

bne x12, x13, TOP      IF .. ID EX ME WB
slli x5, x12, 3        IF .. ID .. EX ME WB

add x6, x10, x5              IF .. ID EX ME WB
ld x7, 0(x6)                 IF .. ID .. EX ME WB

ld x29, 8(x6)                      IF .. ID EX ME WB
sub x30, x7, x29                   IF .. ID .. .. EX ME WB

add x31, x11, x5                         IF .. .. ID EX ME WB
sd x30, 0(x31)                           IF .. .. ID .. EX ME WB

addi x12, x12, 2                                 IF .. ID EX ME WB
bne x12, x13, TOP                                IF .. ID .. EX ME WB

slli x5, x12, 3                                        IF .. ID EX ME WB
add x6, x10, x5                                        IF .. ID .. EX ME WB

ld x7, 0(x6)                                                 IF .. ID EX ME WB
ld x29, 8(x6)                                                IF .. ID .. EX ME WB

sub x30, x7, x29                                                   IF .. ID .. EX ME WB
add x31, x11, x5                                                   IF .. ID .. .. EX ME WB

sd x30, 0(x31)                                                           IF .. .. ID EX ME WB
addi x12, x12, 2                                                         IF .. .. ID .. EX ME WB

bne x12, x13, TOP                                                                  IF .. ID EX ME WB
slli x5, x12, 3                                                                    IF .. ID .. EX ME WB
```

4.31.2 The original code requires 10 cycles/loop on a 1-issue machine (stalls shown below) and 10 cycles/loop on the 2-issue machine. This gives no net speedup. (That's a terrible result considering we nearly doubled the amount of hardware.) We know that the code takes 10 cycles/iteration on the 2-issue machine because the first instruction in loop 1 (The slli) begins execution in cycle 6 and the first instruction in iteration 3 begins execution in cycle 26, so (26–6)/2 = 10.

```
    li x12,0
    jal ENT
TOP:
    slli x5, x12, 3
    add x6, x10, x5
    ld x7, 0(x6)
    ld x29, 8(x6)
    <stall>
    sub x30, x7, x29
    add x31, x11, x5
    sd x30, 0(x31)
    addi x12, x12, 2
ENT:
    bne x12, x13, TOP
```

4.31.3 Here is one possibility:

```
    beqz x13, DONE
    li x12, 0
    jal ENT
TOP:
    slli x5, x12, 3
    add x6, x10, x5
    ld x7, 0(x6)
    ld x29, 8(x6)
    addi x12, x12, 2
    sub x30, x7, x29
    add x31, x11, x5
    sd x30, 0(x31)
ENT:
    bne x12, x13, TOP
DONE:
```

If we switch to a "pointer-based" approach, we can save one cycle/loop.

The code below does this:

```
for (i = 0; i ! = j; i+ = 2) {
  *b = *a - *(a+1);
  b+=2;
  a+=2;
}
```

```
  bez x13, DONE
  li x12, 0
  jal ENT
TOP:
  ld x7, 0(x10)
  ld x29, 8(x10)
  addi x12, x12, 2
  sub x30, x7, x29
  sd x30, 0(x11)
  addi x10, x10, 16
  addi x11, x11, 16
ENT:
  bne x12,x13,TOP
DONE:
```

4.31.4 Here is one possibility:

```
  beqz x13, DONE
  li x12, 0
TOP:
  slli x5, x12, 3
  add x6, x10, x5
  ld x7, 0(x6)
  add x31, x11, x5
  ld x29, 8(x6)
  addi x12, x12, 2
  sub x30, x7, x29
  sd x30, 0(x31)
  bne x12, x13, TOP
DONE:
```

If we switch to a “pointer-based” approach, we can save one cycle/loop.

The code below does this:

```
for (i = 0; i ! = j; i+ = 2) {
   *b = *a - *(a+1);
   b+=2;
   a+=2;
}
   beqz x13, DONE
   li x12, 0
TOP:
   ld x7, 0(x6)
   addi x12, x12, 2
   ld x29, 8(x6)
   addi x6, x6, 16
   sub x30, x7, x29
   sd x30, 0(x31)
   bne x12, x13, TOP
DONE:
```

4.31.5

```
                    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
beqz x13, DONE      IF ID EX ME WB
li x12, 0           IF ID .. EX ME WB

slli x5, x12, 3        IF .. ID EX ME WB
add x6, x10, x5        IF .. ID .. EX ME WB

ld x7, 0(x6)                 IF .. ID EX ME WB
add x31, x11, x5             IF .. ID EX ME WB

ld x29, 8(x6)                      IF ID EX ME WB
addi x12, x12, 2                   IF ID EX ME WB

sub x30, x7, x29                      IF ID .. EX ME WB
sd x30, 0(x31)                        IF ID .. .. EX ME WB

bne x12, x13, TOP                        IF .. .. ID EX ME WB
slli x5, x12, 3                          IF .. .. ID .. EX ME WB

add x6, x10, x5                                   IF .. ID EX ME WB
ld x7, 0(x6)                                      IF .. ID .. EX ME WB

add x31, x11, x5                                        IF .. ID EX ME WB
ld x29, 8(x6)                                           IF .. ID EX ME WB

addi x12, x12, 2                                              IF ID EX ME WB
sub x30, x7, x29                                              IF ID .. EX ME WB

sd x30, 0(x31)                                                   IF .. ID EX ME WB
bne x12, x13, TOP                                                IF .. ID EX ME WB

slli x5, x12, 3                                                        IF ID EX ME WB
add x6, x10, x5                                                        IF ID .. EX ME WB
```

4.31.6 The code from 4.31.3 requires 9 cycles per iteration. The code from 4.31.5 requires 7.5 cycles per iteration. Thus, the speedup is 1.2.

4.31.7 Here is one possibility:

```
      beqz x13, DONE
      li x12, 0
TOP:
      slli x5, x12, 3
      add x6, x10, x5
      add x31, x11, x5
      ld x7, 0(x6)
      ld x29, 8(x6)
      ld x5, 16(x6)
      ld x15, 24(x6)
      addi x12, x12, 4
      sub x30, x7, x29
      sub x14, x5, x15
      sd x30, 0(x31)
      sd x14, 16(x31)
      bne x12, x13, TOP
DONE:
```

4.31.8 Here is one possibility:

```
      beqz x13, DONE
      li x12, 0
      addi x6, x10, 0
TOP:
      ld x7, 0(x6)
      add x31, x11, x5
      ld x29, 8(x6)
      addi x12, x12, 4
      ld x16, 16(x6)
      slli x5, x12, 3
      ld x15, 24(x6)
      sub x30, x7, x29
      sd x30, 0(x31)
      sub x14, x16, x15
      sd x14, 16(x31)
      add x6, x10, x5
      bne x12,x13,TOP
DONE:
```

4.31.9 The code from 4.31.7 requires 13 cycles per unrolled iteration. This is equivalent to 6.5 cycles per original iteration. The code from 4.30.4 requires 7.5 cycles per unrolled iteration. This is equivalent to 3.75 cycles per original iteration. Thus, the speedup is 1.73.

```
                          1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
beqz x13, DONE           IF ID EX ME WB
li x12, 0                IF ID .. EX ME WB

addi x6, x10, 0             IF .. ID EX ME WB
ld x7, 0(x6)                IF .. ID .. EX ME WB

add x31, x11, x5                  IF .. ID EX ME WB
ld x29, 8(x6)                     IF .. ID EX ME WB

addi x12, x12, 4                        IF ID EX ME WB
ld x16, 16(x6)                          IF ID EX ME WB

slli x5, x12, 3                            IF ID EX ME WB
ld x15, 24(x6)                             IF ID EX ME WB

sub x30, x7, x29                              IF ID EX ME WB
sd x30, 0(x31)                                IF ID .. EX ME WB

sub x14, x16, x15                                IF .. ID EX ME WB
sd x14, 16(x31)                                  IF .. ID EX ME WB

add x6, x10, x5                                        IF ID EX ME WB
bne x12,x13,TOP                                        IF ID EX ME WB

ld x7, 0(x6)                                              IF ID EX ME WB
add x31, x11, x5                                          IF ID EX ME WB

ld x29, 8(x6)                                                IF ID EX ME WB
addi x12, x12, 4                                             IF ID EX ME WB

ld x16, 16(x6)                                                  IF ID EX ME WB
slli x5, x12, 3                                                 IF ID EX ME WB

ld x15, 24(x6)                                                     IF ID EX ME WB
sub x30, x7, x29                                                   IF ID EX ME WB

sd x30, 0(x31)                                                        IF ID EX ME WB
sub x14, x16, x15                                                     IF ID .. EX ME WB

sd x14, 16(x31)                                                          IF .. ID EX ME WB
add x6, x10, x5                                                          IF .. ID EX ME WB

bne x12,x13,TOP                                                                IF ID EX ME WB
ld x7, 0(x6)                                                                   IF ID EX ME WB
```

4.31.10 Using the same code as in 4.31.8, the new data path provides no net improvement, because there are no stalls due to structural hazards.

4.32

4.32.1 The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have: 140pJ + 2*70pJ + 60j = 340pJ.

4.32.2 The instruction memory is read for all instructions. Every instruction also results in two register reads (even if only one of those values is actually used). A load instruction results in a memory read and a register write; a store instruction results in a memory write; all other instructions result in at most a single register write. Because the sum of memory read and register write energy is larger than memory write energy, the worst-case instruction is a load instruction. For the energy spent by a load, we have: 140pJ + 2*70pJ + 60pJ + 140pJ = 480pJ.

4.32.3 Instruction memory must be read for every instruction. However, we can avoid reading registers whose values are not going to be used. To do this, we must add RegRead1 and RegRead2 control inputs to the Registers unit to enable or disable each register read. We must generate these control signals quickly to avoid lengthening the clock cycle time. With these new control signals, a load instruction results in only one register read (we still must read the register used to generate the address), so our change saves 70pJ (one register read) per load. This is a savings of 70/480 = 14.6%.

4.32.4 `jal` will benefit, because it need not read any registers at all. I-type instructions will also benefit because they need only read one register. If we add logic to detect x0 as a source register, then instructions such as `beqz` (i.e. `beq x0`, …) and `li` (`addi xn, x0`, …) could benefit as well.

4.32.5 Before the change, the Control unit decodes the instruction while register reads are happening. After the change, the latencies of Control and Register Read cannot be overlapped. This increases the latency of the ID stage and could affect the processor's clock cycle time if the ID stage becomes the longest-latency stage. However, the sum of the latencies for the register read (90ps) and control unit (150ps) are less than the current 250ps cycle time.

4.32.6 If memory is read in every cycle, the value is either needed (for a load instruction), or it does not get past the WB Mux (for a non-load instruction that writes to a register), or it does not get written to any register (all other instructions, including branches and stalls). This change does not affect clock cycle time because the clock cycle time must already allow enough time for memory to be read in the MEM stage. It can affect overall performance if the unused memory reads cause cache misses.

The change also affects energy: A memory read occurs in every cycle instead of only in cycles when a load instruction is in the MEM stage. This increases the energy consumption by 140pJ during 75% of the 250ps clock cycles. This corresponds to a consumption of approximately 0.46 Watts (not counting any energy consumed as a result of cache misses).

4.33

4.33.1 To test for a stuck-at-0 fault on a wire, we need an instruction that puts that wire to a value of 1 and has a different result if the value on the wire is stuck at zero.

If the least significant bit of the write register line is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. To test for this (1) place a value of 10 in `x1`, 35 in `x2`, and 45 in `x3`, then (2) execute `add x3, x1, x1`. The value of `x3` is supposed to be 20. If bit 0 of the Write Register input to the registers unit is stuck at zero, the value is written to `x2` instead, which means that `x2` will be 40 and `x3` will remain at 45.

4.33.2 The test for stuck-at-zero requires an instruction that sets the signal to 1; and the test for stuck-at-1 requires an instruction that sets the signal to 0. Because the signal cannot be both 0 and 1 in the same cycle, we cannot test the same signal simultaneously for stuck-at-0 and stuck-at-1 using only one instruction.

The test for stuck-at-1 is analogous to the stuck-at-0 test: (1) Place a value of 10 in `x1`, 35 in `x2`, and 45 in `x3`, then (2) execute `add x2, x1, x1`. The value of `x2` is supposed to be 20. If bit 0 of the Write Register input to the registers unit is stuck at 1, the value is written to `x3` instead, which means that `x3` will be 40 and `x2` will remain at 35.

4.33.3 The CPU is still usable. The "simplest" solution is to re-write the compiler so that it uses odd-numbered registers only (not that writing compliers is especially simple). We could also write a "translator" that would convert machine code; however, this would be more complicated because the translator would need to detect when two "colliding" registers are used simultaneously and either (1) place one of the values in an unused register, or (2) push that value onto the stack.

4.33.4 To test for this fault, we need an instruction for which MemRead is set to 1, so it has to be ld. The instruction also needs to have branch set to 0, which is the case for ld. Finally, the instruction needs to have a different result MemRead is incorrectly set to 0. For a load, setting MemRead to 0 results in not reading memory. When this happens, the value placed in the register is "random" (whatever happened to be at

the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.

4.33.5 Only R-type instructions set RegRd to 1. Most R-type instructions would fail to detect this error because reads are non-destructive—the erroneous read would simply be ignored. However, suppose we issued this instruction: `add x1, x0, x0`. In this case, if MemRead were incorrectly set to 1, the data memory would attempt to read the value in memory location 0. In many operating systems, address 0 can only be accessed by a process running in protected/kernel mode. If this is the case, then this instruction would cause a segmentation fault in the presence of this error.

5

Solutions

5.1

5.1.1 2.

5.1.2 I, J, and B[I][0].

5.1.3 A[I][J].

5.1.4 I, J, and B[I][0].

5.1.5 A(J, I) and B[I][0].

5.1.6 32,004 with Matlab. 32,008 with C.

The code references 8*8000 = 64,000 integers from matrix A. At two integers per 16-byte block, we need 32,000 blocks.

The code also references the first element in each of eight rows of Matrix B. Matlab stores matrix data in column-major order; therefore, all eight integers are contiguous and fit in four blocks. C stores matrix data in row-major order; therefore, the first element of each row is in a different block.

5.2

5.2.1

Word Address	Binary Address	Tag	Index	Hit/Miss
0x03	0000 0011	0	3	M
0xb4	1011 0100	b	4	M
0x2b	0010 1011	2	b	M
0x02	0000 0010	0	2	M
0xbf	1011 1111	b	f	M
0x58	0101 1000	5	8	M
0xbe	1011 1110	b	e	M
0x0e	0000 1110	0	e	M
0xb5	1011 0101	b	5	M
0x2c	0010 1100	2	c	M
0xba	1011 1010	b	a	M
0xfd	1111 1101	f	d	M

5.2.2

Word Address	Binary Address	Tag	Index	Offset	Hit/Miss
0x03	0000 0011	0	1	1	M
0xb4	1011 0100	b	2	0	M
0x2b	0010 1011	2	5	1	M
0x02	0000 0010	0	1	0	H
0xbf	1011 1111	b	7	1	M
0x58	0101 1000	5	4	0	M
0xbe	1011 1110	b	6	0	H
0x0e	0000 1110	0	7	0	M
0xb5	1011 0101	b	2	1	H
0x2c	0010 1100	2	6	0	M
0xba	1011 1010	b	5	0	M
0xfd	1111 1101	f	6	1	M

5.2.3

Word Address	Binary Address	Tag	Cache 1		Cache 2		Cache 3	
			Index	Hit/miss	Index	Hit/miss	Index	Hit/miss
0x03	0000 0011	0x00	3	M	1	M	0	M
0xb4	1011 0100	0x16	4	M	2	M	1	M
0x2b	0010 1011	0x05	3	M	1	M	0	M
0x02	0000 0010	0x00	2	M	1	M	0	M
0xbf	1011 1111	0x17	7	M	3	M	1	M
0x58	0101 1000	0x0b	0	M	0	M	0	M
0xbe	1011 1110	0x17	6	M	3	H	1	H
0x0e	0000 1110	0x01	6	M	3	M	1	M
0xb5	1011 0101	0x16	5	M	2	H	1	M
0x2c	0010 1100	0x05	4	M	2	M	1	M
0xba	1011 1010	0x17	2	M	1	M	0	M
0xfd	1111 1101	0x1F	5	M	2	M	1	M

Cache 1 miss rate = 100%

Cache 1 total cycles = $12 \times 25 + 12 \times 2 = 324$

Cache 2 miss rate = 10/12 = 83%

Cache 2 total cycles = $10 \times 25 + 12 \times 3 = 286$

Cache 3 miss rate = 11/12 = 92%

Cache 3 total cycles = $11 \times 25 + 12 \times 5 = 335$

Cache 2 provides the best performance.

5.3

5.3.1 Total size is 364,544 bits = 45,568 bytes

Each word is 8 bytes; each block contains two words; thus, each block contains 16 = 2^4 bytes.

The cache contains 32KiB = 2^15 bytes of data. Thus, it has 2^15/2^4 = 2^11 lines of data.

Each 64-bit address is divided into: (1) a 3-bit word offset, (2) a 1-bit block offset, (3) an 11-bit index (because there are 2^11 lines), and (4) a 49-bit tag (64 − 3 − 1 − 11 = 49).

The cache is composed of: 2^15 * 8 bits of data + 2^11*49 bits of tag + 2^11*1 valid bits = 364,544 bits.

5.3.2 549,376 bits = 68,672 bytes. This is a 51% increase.

Each word is 8 bytes; each block contains 16 words; thus, each block contains 128 = 2^7 bytes.

The cache contains 64KiB = 2^16 bytes of data. Thus, it has 2^16/2^7 = 2^9 lines of data.

Each 64-bit address is divided into: (1) a 3-bit word offset, (2) a 4-bit block offset, (3) a 9-bit index (because there are 2^9 lines), and (4) a 48-bit tag (64 − 3 − 4 − 9 = 48).

The cache is composed of: 2^16 * 8 bits of data + 2^9*48 bits of tag + 2^9*1 valid bits = 549,376 bits

5.3.3 The larger block size may require an increased hit time and an increased miss penalty than the original cache. The fewer number of blocks may cause a higher conflict miss rate than the original cache.

5.3.4 Associative caches are designed to reduce the rate of conflict misses. As such, a sequence of read requests with the same 12-bit index field but a different tag field will generate many misses. For the cache described above, the sequence 0, 32768, 0, 32768, 0, 32768, …, would miss on every access, while a two-way set associate cache with LRU replacement, even one with a significantly smaller overall capacity, would hit on every access after the first two.

5.4 Yes it is possible. To implement a direct-mapped cache, we need only a function that will take an address as input and produce a 10-bit output. Although it is possible to implement a cache in this manner, it is not clear that such an implementation will be beneficial. (1) The cache would require a larger tag and (2) there would likely be more conflict misses.

5.5

5.5.1 Each cache block consists of four 8-byte words. The total offset is 5 bits. Three of those 5 bits is the word offset (the offset into an 8-byte word). The remaining two bits are the block offset. Two bits allows us to enumerate 2^2 = 4 words.

5.5.2 There are five index bits. This tells us there are 2^5 = 32 lines in the cache.

5.5.3 The ratio is 1.21. The cache stores a total of 32 lines * 4 words/block * 8 bytes/word = 1024 bytes = 8192 bits.

In addition to the data, each line contains 54 tag bits and 1 valid bit. Thus, the total bits required = 8192 + 54*32 + 1*32 = 9952 bits.

5.5.4

Byte Address	Binary Address	Tag	Index	Offset	Hit/Miss	Bytes Replaced
0x00	0000 0000 0000	0x0	0x00	0x00	M	
0x04	0000 0000 0100	0x0	0x00	0x04	H	
0x10	0000 0001 0000	0x0	0x00	0x10	H	
0x84	0000 1000 0100	0x0	0x04	0x04	M	
0xe8	0000 1110 1000	0x0	0x07	0x08	M	
0xa0	0000 1010 0000	0x0	0x05	0x00	M	
0x400	0100 0000 0000	0x1	0x00	0x00	M	0x00-0x1F
0x1e	0000 0001 1110	0x0	0x00	0x1e	M	0x400-0x41F
0x8c	0000 1000 1100	0x0	0x04	0x0c	H	
0xc1c	1100 0001 1100	0x3	0x00	0x1c	M	0x00-0x1F
0xb4	0000 1011 0100	0x0	0x05	0x14	H	
0x884	1000 1000 0100	0x2	0x04	0x04	M	0x80-0x9f

5.5.5 4/12 = 33%.

5.5.6
```
<index, tag, data>
<0, 3, Mem[0xC00]-Mem[0xC1F]>
<4, 2, Mem[0x880]-Mem[0x89f]>
<5, 0, Mem[0x0A0]-Mem[0x0Bf]>
<7, 0, Mem[0x0e0]-Mem[0x0ff]>
```

5.6

5.6.1 The L1 cache has a low write miss penalty while the L2 cache has a high write miss penalty. A write buffer between the L1 and L2 cache would hide the write miss latency of the L2 cache. The L2 cache would benefit from write buffers when replacing a dirty block, since the new block would be read in before the dirty block is physically written to memory.

5.6.2 On an L1 write miss, the word is written directly to L2 without bringing its block into the L1 cache. If this results in an L2 miss, its block must be brought into the L2 cache, possibly replacing a dirty block, which must first be written to memory.

5.6.3 After an L1 write miss, the block will reside in L2 but not in L1. A subsequent read miss on the same block will require that the block in L2 be written back to memory, transferred to L1, and invalidated in L2.

5.7

5.7.1 When the CPI is 2, there are, on average, 0.5 instruction accesses per cycle. 0.3% of these instruction accesses cause a cache miss (and subsequent memory request). Assuming each miss requests one block, instruction accesses generate an average of 0.5*.003*64 = 0.096 bytes/cycle of read traffic.

25% of instructions generate a read request. 2% of these generate a cache miss; thus, read misses generate an average of 0.5*0.25*0.02*64 = 0.16 bytes/cycle of read traffic.

10% of instructions generate a write request. 2% of these generate a cache miss. Because the cache is a write-through cache, only one word (8 bytes) must be written back to memory; but, every write is written through to memory (not just the cache misses). Thus, write misses generate an average of 0.5*0.1*8 = 0.4 bytes/cycle of write traffic. Because the cache is a write-allocate cache, a write miss also makes a read request to RAM. Thus, write misses require an average of 0.5*0.1*0.02*64 = 0.064 bytes/cycle of read traffic.

Hence: The total read bandwidth = 0.096 + 0.16 + 0.064 = 0.32 bytes/cycle, and the total write bandwidth is 0.4 bytes/cycle.

5.7.2 The instruction and data read bandwidth requirement is the same as in 5.4.4.

With a write-back cache, data are only written to memory on a cache miss. But, it is written on every cache miss (both read and write), because any line could have dirty data when evicted, even if the eviction is caused by a read request. Thus, the data write bandwidth requirement becomes 0.5*(0.25 + 0.1)*0.02*0.3*64 = 0.0672 bytes/cycle.

5.8

5.8.1 The addresses are given as word addresses; each 32-bit block contains four words. Thus, every fourth access will be a miss (i.e., a miss rate of 1/4). All misses are compulsory misses. The miss rate is not sensitive to the size of the cache or the size of the working set. It is, however, sensitive to the access pattern and block size.

5.8.2 The miss rates are 1/2, 1/8, and 1/16, respectively. The workload is exploiting spatial locality.

5.8.3 In this case the miss rate is 0: The pre-fetch buffer always has the next request ready.

5.9

5.9.1 AMAT for B = 8: $0.040 \times (20 \times 8) = 6.40$

AMAT for B = 16: $0.030 \times (20 \times 16) = 9.60$

AMAT for B = 32: $0.020 \times (20 \times 32) = 12.80$

AMAT for B = 64: $0.015 \times (20 \times 64) = 19.20$

AMAT for B = 128: $0.010 \times (20 \times 128) = 25.60$

B = 8 is optimal.

5.9.2 AMAT for B = 8: $0.040 \times (24 + 8) = 1.28$

AMAT for B = 16: $0.030 \times (24 + 16) = 1.20$

AMAT for B = 32: $0.020 \times (24 + 32) = 1.12$

AMAT for B = 64: $0.015 \times (24 + 64) = 1.32$

AMAT for B = 128: $0.010 \times (24 + 128) = 1.52$

B = 32 is optimal

5.9.3 B = 128 is optimal: Minimizing the miss rate minimizes the total miss latency.

5.10

5.10.1

P1	1.515 GHz
P2	1.11 GHz

5.10.2

P1	6.31 ns	9.56 cycles
P2	5.11 ns	5.68 cycles

For P1 all memory accesses require at least one cycle (to access L1). 8% of memory accesses additionally require a 70 ns access to main memory. This is 70/0.66 = 106.06 cycles. However, we can't divide cycles; therefore, we must round up to 107 cycles. Thus, the Average Memory Access time is 1 + 0.08*107 = 9.56 cycles, or 6.31 ps.

For P2, a main memory access takes 70 ns. This is 70/0.66 = 77.78 cycles. Because we can't divide cycles, we must round up to 78 cycles. Thus the Average Memory Access time is 1 + 0.06*78 = 5.68 cycles, or 6.11 ps.

5.10.3

P1	12.64 CPI	8.34 ns per inst
P2	7.36 CPI	6.63 ns per inst

For P1, every instruction requires at least one cycle. In addition, 8% of all instructions miss in the instruction cache and incur a 107-cycle delay. Furthermore, 36% of the instructions are data accesses. 8% of these 36% are cache misses, which adds an additional 107 cycles.

1 + .08*107 + .36*.08*107 = 12.64

With a clock cycle of 0.66 ps, each instruction requires 8.34 ns.

Using the same logic, we can see that P2 has a CPI of 7.36 and an average of only 6.63 ns/instruction.

5.10.4

AMAT = 9.85 cycles	Worse

An L2 access requires nine cycles (5.62/0.66 rounded up to the next integer).

All memory accesses require at least one cycle. 8% of memory accesses miss in the L1 cache and make an L2 access, which takes nine cycles. 95% of all L2 access are misses and require a 107 cycle memory lookup.

1 + .08[9 + 0.95*107] = 9.85

5.10.5

13.04

Notice that we can compute the answer to 5.6.3 as follows: AMAT + %memory * (AMAT-1).

Using this formula, we see that the CPI for P1 with an L2 cache is 9.85 * 0.36*8.85 = 13.04

5.10.6 Because the clock cycle time and percentage of memory instructions is the same for both versions of P1, it is sufficient to focus on AMAT. We want

AMAT with L2 < AMAT with L1 only

1 + 0.08[9 + m*107] < 9.56

This happens when m< .916.

5.10.7 We want P1's average time per instruction to be less than 6.63 ns. This means that we want

(CPI_P1 * 0.66) < 6.63. Thus, we need CPI_P1 < 10.05

CPI_P1 = AMAT_P1 + 0.36(AMAT_P1 – 1)

Thus, we want

AMAT_P1+ 0.36(AMAT_P1-1) < 10.05

This happens when AMAT_P1< 7.65.

Finally, we solve for

1+ 0.08[9+ m*107] < 7.65

and find that

m< 0.693

This miss rate can be at most 69.3%.

5.11

5.11.1 Each line in the cache will have a total of six blocks (two in each of three ways). There will be a total of 48/6 = 8 lines.

5.11.2 T(x) is the tag at index x.

Word Address	Binary Address	Tag	Index	Offset	Hit/Miss	Way 0	Way 1	Way 2
0x03	0000 0011	0x0	1	1	M	T(1)=0		
0xb4	1011 0100	0xb	2	0	M	T(1)=0 T(2)=b		
0x2b	0010 1011	0x2	5	1	M	T(1)=0 T(2)=b T(5)=2		
0x02	0000 0010	0x0	1	0	H	T(1)=0 T(2)=b T(5)=2		
0xbe	1011 1110	0xb	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b		
0x58	0101 1000	0x5	4	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5		
0xbf	1011 1111	0xb	7	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5		
0x0e	0000 1110	0x0	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	
0x1f	0001 1111	0x1	7	1	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xb5	1011 0101	0xb	2	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xbf	1011 1111	0xb	7	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xba	1011 1010	0xb	5	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=1
0x2e	0010 1110	0x2	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=1
0xce	1100 1110	0xc	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=c

5.11.3 No solution given.

5.11.4 Because this cache is fully associative and has one-word blocks, there is no index and no offset. Consequently, the word address is equivalent to the tag.

Word Address	Binary Address	Tag	Hit/Miss	Contents
0x03	0000 0011	0x03	M	3
0xb4	1011 0100	0xb4	M	3, b4
0x2b	0010 1011	0x2b	M	3, b4, 2b
0x02	0000 0010	0x02	M	3, b4, 2b, 2
0xbe	1011 1110	0xbe	M	3, b4, 2b, 2, be
0x58	0101 1000	0x58	M	3, b4, 2b, 2, be, 58
0xbf	1011 1111	0xbf	M	3, b4, 2b, 2, be, 58, bf
0x0e	0000 1110	0x0e	M	3, b4, 2b, 2, be, 58, bf, e
0x1f	0001 1111	0x1f	M	b4, 2b, 2, be, 58, bf, e, 1f
0xb5	1011 0101	0xb5	M	2b, 2, be, 58, bf, e, 1f, b5
0xbf	1011 1111	0xbf	H	2b, 2, be, 58, e, 1f, b5, bf
0xba	1011 1010	0xba	M	2, be, 58, e, 1f, b5, bf, ba
0x2e	0010 1110	0x2e	M	be, 58, e, 1f, b5, bf, ba, 2e
0xce	1100 1110	0xce	M	58, e, 1f, b5, bf, ba, 2e, ce

5.11.5 No solution given.

5.11.6 Because this cache is fully associative, there is no index. (Contents shown in the order the data were accessed. Order does not imply physical location.)

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[b4,b5], [2a,2b], [2,3]
0xbe	1011 1110	0x5f	0	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x58	0101 1000	0x2c	0	M	[2a,2b], [2,3], [be, bf], [58, 59]
0xbf	1011 1111	0x5f	1	H	[2a,2b], [2,3], [58, 59], [be, bf]
0x0e	0000 1110	0x07	0	M	[2,3], [58, 59], [be, bf], [e,f]
0x1f	0001 1111	0x0f	1	M	[58, 59], [be, bf], [e,f], [1e,1f]
0xb5	1011 0101	0x5a	1	M	[be, bf], [e,f], [1e,1f], [b4, b5]
0xbf	1011 1111	0x5f	1	H	[e,f], [1e,1f], [b4, b5], [be, bf]
0xba	1011 1010	0x5d	0	M	[1e,1f], [b4, b5], [be, bf], [ba, bb]
0x2e	0010 1110	0x17	0	M	[b4, b5], [be, bf], [ba, bb], [2e, 2f]
0xce	1100 1110	0x67	0	M	[be, bf], [ba, bb], [2e, 2f], [ce,cf]

5.11.7 (Contents shown in the order the data were accessed. Order does not imply physical location.)

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[b4,b5], [2a,2b], [2,3]
0xbe	1011 1110	0x5f	0	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x58	0101 1000	0x2c	0	M	[b4,b5], [2a,2b], [2,3], [58, 59]
0xbf	1011 1111	0x5f	1	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x0e	0000 1110	0x07	0	M	[b4,b5], [2a,2b], [2,3], [e, f]
0x1f	0001 1111	0x0f	1	M	[b4,b5], [2a,2b], [2,3], [1e, 1f]
0xb5	1011 0101	0x5a	1	H	[2a,2b], [2,3], [1e, 1f], [b4,b5]
0xbf	1011 1111	0x5f	1	M	[2a,2b], [2,3], [1e, 1f], [be, bf]
0xba	1011 1010	0x5d	0	M	[2a,2b], [2,3], [1e, 1f], [ba, bb]
0x2e	0010 1110	0x17	0	M	[2a,2b], [2,3], [1e, 1f], [2e, 2f]
0xce	1100 1110	0x67	0	M	[2a,2b], [2,3], [1e, 1f],, [ce, cf]

5.11.8 Because this cache is fully associative, there is no index.

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[2,3], [b4,b5], [2a,2b]
0xbe	1011 1110	0x5f	0	M	[2,3], [b4,b5], [2a,2b], [be, bf]
0x58	0101 1000	0x2c	0	M	[58,59], [b4,b5], [2a,2b], [be, bf]
0xbf	1011 1111	0x5f	1	H	[58,59], [b4,b5], [2a,2b], [be, bf]
0x0e	0000 1110	0x07	0	M	[e,f], [b4,b5], [2a,2b], [be, bf]
0x1f	0001 1111	0x0f	1	M	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xb5	1011 0101	0x5a	1	H	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xbf	1011 1111	0x5f	1	H	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xba	1011 1010	0x5d	0	M	[1e,1f], [b4,b5], [ba,bb], [be, bf]
0x2e	0010 1110	0x17	0	M	[1e,1f], [b4,b5], [2e,2f], [be, bf]
0xce	1100 1110	0x67	0	M	[1e,1f], [b4,b5], [ce,cf], [be, bf]

5.12

5.12.1 Standard memory time: Each cycle on a 2-Ghz machine takes 0.5 ps. Thus, a main memory access requires 100/0.5 = 200 cycles.

- L1 only: 1.5 + 0.07*200 = 15.5
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 200) = 2.83
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 200) = 3.67.

Doubled memory access time (thus, a main memory access requires 400 cycles)

- L1 only: 1.5 + 0.07*400 = 29.5 (90% increase)
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 400) = 3.32 (17% increase)
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 400) = 3.88 (5% increase).

5.12.2 $1.5 = 0.07 \times (12 + 0.035 \times (50 + 0.013 \times 100)) = 2.47$

Adding the L3 cache does reduce the overall memory access time, which is the main advantage of having an L3 cache. The disadvantage is that the L3 cache takes real estate away from having other types of resources, such as functional units.

5.12.3 No size will achieve the performance goal.

We want the CPI of the CPU with an external L2 cache to be at most 2.83. Let x be the necessary miss rate.

1.5 + 0.07*(50 + x*200) < 2.83

Solving for x gives that $x < -0.155$. This means that even if the miss rate of the L2 cache was 0, a 50-ns access time gives a CPI of 1.5 + 0.07*(50 + 0*200) = 5, which is greater than the 2.83 given by the on-chip L2 caches. As such, no size will achieve the performance goal.

5.13

5.13.1

3 years and 1 day	1096 days	26304 hours

5.13.2

1095/1096 = 99.90875912%

5.13.3 Availability approaches 1.0. With the emergence of inexpensive drives, having a nearly 0 replacement time *for hardware* is quite feasible. However, replacing file systems and other data can take significant time. Although a drive manufacturer will not include this time in their statistics, it is certainly a part of replacing a disk.

5.13.4 MTTR becomes the dominant factor in determining availability. However, availability would be quite high if MTTF also grew measurably. If MTTF is 1000 times MTTR, it the specific value of MTTR is not significant.

5.14

5.14.1 9. For SEC, we need to find minimum p such that $2^p >= p + d + 1$ and then add one. That gives us p = 8. We then need to add one more bit for SEC/DED.

5.14.2 The (72,64) code described in the chapter requires an overhead of 8/64 = 12.5% additional bits to tolerate the loss of any single bit within 72 bits, providing a protection rate of 1.4%. The (137,128) code from part a requires an overhead of 9/128 = 7.0% additional bits to tolerate the loss of any single bit within 137 bits, providing a protection rate of 0.73%. The cost/performance of both codes is as follows:

(72,64) code = >12.5/1.4 = 8.9

(136,128) code = >7.0/0.73 = 9.6

The (72,64) code has better cost/performance ratio.

5.14.3 Using the bit numbering from Section 5.5, bit 8 is in error so the value would be corrected to 0x365.

5.15 Instructors can change the disk latency, transfer rate and optimal page size for more variants. Refer to Jim Gray's paper on the 5-minute rule 10 years later.

5.15.1 32 KB.

To solve this problem, I used the following gnuplot command and looked for the maximum.

```
plot [16:128] log((x*1024/128) *0.7)/(log(2)*(10 + 0.1*x))
```

5.15.2 Still 32 KB. (Modify the gnuplot command above by changing 0.7 to 0.5.)

5.15.3 64 KB. Because the disk bandwidth grows much faster than seek latency, future paging cost will be more close to constant, thus favoring larger pages.

1987/1997/2007: 205/267/308 seconds. (or roughly five minutes).

1987/1997/2007: 51/533/4935 seconds. (or 10 times longer for every 10 years).

5.15.4 (1) DRAM cost/MB scaling trend dramatically slows down; or (2) disk $/access/sec dramatically increase. (2) is more likely to happen due to the emerging flash technology.

5.16

5.16.1

Address	Virtual Page	TLB H/M	TLB		
			Valid	Tag	Physical Page
4669 0x123d	1	TLB miss PT hit PF	1	b	12
			1	7	4
			1	3	6
			1 (last access 0)	1	13
2227 0x08b3	0	TLB miss PT hit	1 (last access 1)	0	5
			1	7	4
			1	3	6
			1 (last access 0)	1	13
13916 0x365c	3	TLB miss PT hit	1 (last access 1)	0	5
			1	7	4
			1 (last access 2)	3	6
			1 (last access 0)	1	13
34587 0x871b	8	TLB miss PT hit PF	1 (last access 1)	0	5
			1 (last access 3)	8	14
			1 (last access 2)	3	6
			1 (last access 0)	1	13
48870 0xbee6	b	TLB miss PT hit	1 (last access 1)	0	5
			1 (last access 3)	8	14
			1 (last access 2)	3	6
			1 (last access 4)	b	12
12608 0x3140	3	TLB miss PT hit	1 (last access 1)	0	5
			1 (last access 3)	8	14
			1 (last access 5)	3	6
			1 (last access 4)	b	12
49225 0xc040	c	TLB miss PT hit PF	1 (last access 6)	c	15
			1 (last access 3)	8	14
			1 (last access 5)	3	6
			1 (last access 4)	b	12

5.16.2

Address	Virtual Page	TLB H/M	TLB		
			Valid	Tag	Physical Page
4669 0x123d	1	TLB miss PT hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 0)	0	5
2227 0x08b3	0	TLB hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 1)	0	5
13916 0x365c	0	TLB hit PT hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 2)	0	5
34587 0x871b	2	TLB miss PT hit PF	1 (last access 3)	2	13
			1	7	4
			1	3	6
			2	0	5
48870 0xbee6	2	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1	3	6
			1 (last access 2)	0	5
12608 0x3140	0	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1	3	6
			5	0	5
49225 0xc040	3	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1 (last access 6)	3	6
			1 (last access 5)	0	5

A larger page size reduces the TLB miss rate but can lead to higher fragmentation and lower utilization of the physical memory.

5.16.3 Two-way set associative

Address	Virtual Page	Tag	Index	TLB H/M	TLB			
					Valid	Tag	Physical Page	Index
4669 0x123d	1	0	1	TLB miss PT hit PF	1	b	12	0
					1	7	4	1
					1	3	6	0
					1 (last access 0)	0	13	1
2227 0x08b3	0	0	0	TLB miss PT hit	1 (last access 1)	0	5	0
					1	7	4	1
					1	3	6	0
					1 (last access 0)	0	13	1
13916 0x365c	3	1	1	TLB miss PT hit	1 (last access 1)	0	5	0
					1 (last access 2)	1	6	1
					1	3	6	0
					1 (last access 0)	1	13	1
34587 0x871b	8	4	0	TLB miss PT hit PF	1 (last access 1)	0	5	0
					1 (last access 2)	1	6	1
					1 (last access 3)	4	14	0
					1 (last access 0)	1	13	1
48870 0xbee6	b	5	1	TLB miss PT hit	1 (last access 1)	0	5	0
					1 (last access 2)	1	6	1
					1 (last access 3)	4	14	0
					1 (last access 4)	5	12	1
12608 0x3140	3	1	1	TLB hit PT hit	1 (last access 1)	0	5	0
					1 (last access 5)	1	6	1
					1 (last access 3)	4	14	0
					1 (last access 4)	5	12	1
49225 0xc049	c	6	0	TLB miss PT miss PF	1 (last access 6)	6	15	0
					1 (last access 5)	1	6	1
					1 (last access 3)	4	14	0
					1 (last access 4)	5	12	1

5.16.4 Direct mapped

Address	Virtual Page	Tag	Index	TLB H/M	TLB			
					Valid	Tag	Physical Page	Index
4669 0x123d	1	0	1	TLB miss PT hit PF	1	b	12	0
					1	0	13	1
					1	3	6	2
					0	4	9	3
2227 0x08b3	0	0	0	TLB miss PT hit	1	0	5	0
					1	0	13	1
					1	3	6	2
					0	4	9	3
13916 0x365c	3	0	3	TLB miss PT hit	1	0	5	0
					1	0	13	1
					1	3	6	2
					1	0	6	3
34587 0x871b	8	2	0	TLB miss PT hit PF	1	2	14	0
					1	0	13	1
					1	3	6	2
					1	0	6	3
48870 0xbee6	b	2	3	TLB miss PT hit	1	2	14	0
					1	0	13	1
					1	3	6	2
					1	2	12	3
12608 0x3140	3	0	3	TLB hit PT hit	1	2	14	0
					1	0	13	1
					1	3	6	2
					1	0	6	3
49225 0xc049	c	3	0	TLB miss PT miss PF	1	3	15	0
					1	0	13	1
					1	3	6	2
					1	0	6	3

5.16.5 Without a TLB, almost every memory access would require two accesses to RAM: An access to the page table, followed by an access to the requested data.

5.17

5.17.1 The tag size is 32–log2(8192) = 32–13 = 19 bits. All five page tables would require 5 × (2^19 × 4) bytes = 10 MB.

5.17.2 In the two-level approach, the 2^19 page table entries are divided into 256 segments that are allocated on demand. Each of the second-level tables contains 2^(19 – 8) = 2048 entries, requiring 2048 × 4 = 8 KB each and covering 2048 × 8 KB = 16 MB (2^24) of the virtual address space.

If we assume that "half the memory" means 2^31 bytes, then the minimum amount of memory required for the second-level tables would be 5 × (2^31/2^24)*8 KB = 5 MB. The first-level tables would require an additional 5 × 128 × 6 bytes = 3840 bytes.

The maximum amount would be if all 1st-level segments were activated, requiring the use of all 256 segments in each application. This would require 5 × 256 × 8 KB = 10 MB for the second-level tables and 7680 bytes for the first-level tables.

5.17.3 The page index is 13 bits (address bits 12 down to 0).

A 16 KB direct-mapped cache with two 64-bit words per block would have 16-byte blocks and thus 16 KB/16 bytes = 1024 blocks. Thus, it would have 10 index bits and 4 offset bits and the index would extend outside of the page index.

The designer could increase the cache's associativity. This would reduce the number of index bits so that the cache's index fits completely inside the page index.

5.18

5.18.1 Worst case is 2^(43 − 12) = 2^31 entries, requiring 2^(31) × 4 bytes = 2^33 = 8 GB.

5.18.2 With only two levels, the designer can select the size of each page table segment. In a multi-level scheme, reading a PTE requires an access to each level of the table.

5.18.3 Yes, if segment table entries are assumed to be the physical page numbers of segment pages, and one bit is reserved as the valid bit, then each one has an effective reach of (2^{31}) * 4KiB = 8TiB, which is more than enough to cover the physical address space of the machine (16 GiB).

5.18.4 Each page table level contains 4KiB/4B = 1024 entries, and so translates log2(1024) = 10 bits of virtual address. Using 43-bit virtual addresses and 4KiB pages, we need ceil((43 − 12)/10) = 4 levels of translation.

5.18.5 In an inverted page table, the number of PTEs can be reduced to the size of the hash table plus the cost of collisions. In this case, serving a TLB miss requires an extra reference to compare the tag or tags stored in the hash table.

5.19

5.19.1 It would be invalid if it was paged out to disk.

5.19.2 A write to page 30 would generate a TLB miss. Software-managed TLBs are faster in cases where the software can pre-fetch TLB entries.

5.19.3 When an instruction writes to VA page 200, an interrupt would be generated because the page is marked as read only.

5.20

5.20.1 There are no hits.

5.20.2 Direct mapped

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0
M	M	M	M	M	M	M	M	H	H	M	M	M	M	H	H	M

5.20.3 Answers will vary.

5.20.4 MRU is an optimal policy.

5.20.5 The best block to evict is the one that will cause the fewest misses in the future. Unfortunately, a cache controller cannot know the future! Our best alternative is to make a good prediction.

5.20.6 If you knew that an address had limited temporal locality and would conflict with another block in the cache, choosing not to cache it could improve the miss rate. On the other hand, you could worsen the miss rate by choosing poorly which addresses to cache.

5.21

5.21.1 CPI = 1.5 + 120/10000 × (15 + 175) = 3.78

If VMM overhead doubles = >CPI = 1.5 + 120/10000 × (15 + 350) = 5.88

If VMM overhead halves = >CPI = 1.5 + 120/10000 × (15 + 87.5) = 2.73

The CPI of a machine running on native hardware is 1.5 + 120/10000*15 = 1.68. To keep the performance degradation to 10%, we need

1.5 + 120/10000*(15 + x) < 1.1*1.68

Solving for x shows that a trap to the VMM can take at most 14 cycles.

5.21.2 Non-virtualized CPI = 1.5 + 120/10000 × 15 + 30/10000 × 1100 = 4.98

Virtualized CPI = 1.5 + 120/10000 × (15 + 175) + 30/10000 × (1100 + 175) = 7.60

Non-virtualized CPI with half I/O = 1.5 + 120/10000 × 15 + 15/10000 × 1100 = 3.33

Virtualized CPI with half I/O = 1.5 + 120/10000 × (15 + 175) + 15/10000 × (1100 + 175) = 5.69.

5.22 Virtual memory aims to provide each application with the illusion of the entire address space of the machine. Virtual machines aim to provide each operating system with the illusion of having the entire machine at its disposal. Thus they both serve very similar goals, and offer benefits such as increased security. Virtual memory can allow for many applications running in the same memory space to not have to manage keeping their memory separate.

5.23 Emulating a different ISA requires specific handling of that ISA's API. Each ISA has specific behaviors that will happen upon instruction execution, interrupts, trapping to kernel mode, etc. that therefore must be emulated. This can require many

more instructions to be executed to emulate each instruction than was originally necessary in the target ISA. This can cause a large performance degradation and make it difficult to properly communicate with external devices. An emulated system can potentially run faster than on its native ISA if the emulated code can be dynamically examined and optimized. For example, if the underlying machine's ISA has a single instruction that can handle the execution of several of the emulated system's instructions, then potentially the number of instructions executed can be reduced. This is similar to the recent Intel processors that do micro-op fusion, allowing several instructions to be handled by fewer instructions.

5.24

5.24.1 The cache should be able to satisfy the request since it is otherwise idle when the write buffer is writing back to memory. If the cache is not able to satisfy hits while writing back from the write buffer, the cache will perform little or no better than the cache without the write buffer, since requests will still be serialized behind writebacks.

5.24.2 Unfortunately, the cache will have to wait until the writeback is complete since the memory channel is occupied. Once the memory channel is free, the cache is able to issue the read request to satisfy the miss.

5.24.3 Correct solutions should exhibit the following features:

1. The memory read should come before memory writes.
2. The cache should signal "Ready" to the processor before completing the write.

5.25

5.25.1 There are six possible orderings for these instructions.

Ordering 1:

P1	P2
X[0]++;	
X[1] = 3;	
	X[0]=5
	X[1] += 2;

Results: (5,5)

Ordering 2:

P1	P2
X[0]++;	
	X[0]=5
X[1] = 3;	
	X[1] += 2;

Results: (5,5)

Ordering 3:

P1	P2
	X[0]=5
X[0]++;	
	X[1] += 2;
X[1] = 3;	

Results: (6,3)

Ordering 4:

P1	P2
X[0]++;	
	X[0]=5
	X[1] += 2;
X[1] = 3;	

Results: (5,3)

Ordering 5:

P1	P2
	X[0]=5
X[0]++;	
X[1] = 3;	
	X[1] += 2;

Results: (6,5)

Ordering 6:

P1	P2
	X[0] = 5
	X[1] += 2;
X[0]++;	
X[1] = 3;	

(6,3)

If coherency isn't ensured:

P2's operations take precedence over P1's: (5,2).

5.25.2 Direct mapped

P1	P1 cache status/action	P2	P2 cache status/action
		X[0]=5	invalidate X on other caches, read X in exclusive state, write X block in cache
		X[1] += 2;	read and write X block in cache
X[0]++;	read value of X into cache		X block enters shared state
	send invalidate message write X block in cache		X block is invalided
X[1] = 3;	write X block in cache		

5.25.3 Best case:

Orderings 1 and 6 above, which require only two total misses.

Worst case:

Orderings 2 and 3 above, which require four total cache misses.

5.25.4

Ordering 1:

P1	P2
A = 1	
B = 2	
A += 2;	
B++;	
	C = B
	D = A

Result: (3,3)

Ordering 2:

P1	P2
A = 1	
B = 2	
A += 2;	
	C = B
B++;	
	D = A

Result: (2,3)

Ordering 3:

P1	P2
A = 1	
B = 2	
	C = B
A += 2;	
B++;	
	D = A

Result: (2,3)

Ordering 4:

P1	P2
A = 1	
	C = B
B = 2	
A += 2;	
B++;	
	D = A

Result: (0,3)

Ordering 5:

P1	P2
	C = B
A = 1	
B = 2	
A += 2;	
B++;	
	D = A

Result: (0,3)

Ordering 6:

P1	P2
A = 1	
B = 2	
A += 2;	
	C = B
	D = A
B++;	

Result: (2,3)

Ordering 7:

P1	P2
A = 1	
B = 2	
	C = B
A += 2;	
	D = A
B++;	

Result: (2,3)

Ordering 8:

P1	P2
A = 1	
	C = B
B = 2	
A += 2;	
	D = A
B++;	

Result: (0,3)

Ordering 9:

P1	P2
	C = B
A = 1	
B = 2	
A += 2;	
	D = A
B++;	

Result: (0,3)

Ordering 10:

P1	P2
A = 1	
B = 2	
	C = B
	D = A
A += 2;	
B++;	

Result: (2,1)

Ordering 11:

P1	P2
A = 1	
	C = B
B = 2	
	D = A
A += 2;	
B++;	

Result: (0,1)

Ordering 12:

P1	P2
	C = B
A = 1	
B = 2	
	D = A
A += 2;	
B++;	

Result: (0,1)

Ordering 13:

P1	P2
A = 1	
	C = B
	D = A
B = 2	
A += 2;	
B++;	

Result: (0,1)

Ordering 14:

P1	P2
	C = B
A = 1	
	D = A
B = 2	
A += 2;	
B++;	

Result: (0,1)

Ordering 15:

P1	P2
	C = B
	D = A
A = 1	
B = 2	
A += 2;	
B++;	

Result: (0,0)

5.25.5 Assume B = 0 is seen by P2 but not preceding A = 1

Result: (2,0).

5.25.6 Write back is simpler than write through, since it facilitates the use of exclusive access blocks and lowers the frequency of invalidates. It prevents the use of write-broadcasts, but this is a more complex protocol.

The allocation policy has little effect on the protocol.

5.26

5.26.1 Benchmark A

AMATprivate = 1 + 0.03*[5 + 0.1*180] = 1.69

AMATshared = 1 + 0.03*[20 + 0.04*180] = 1.82

Benchmark B

AMATprivate = 1+ 0.03*[5 + 0.02*180] = 1.26

AMATshared = 1+ 0.03*[20 + 0.01*180] = 1.65

Private cache is superior for both benchmarks.

5.26.2 In a private cache system, the first link of the chip is the link from the private L2 caches to memory. Thus, the memory latency doubles to 360. In a shared cache system, the first link off the chip is the link to the L2 cache. Thus, in this case, the shared cache latency doubles to 40.

Benchmark A

$AMAT_{private} = 1 + .03*[5 + .1*360] = 2.23$

$AMAT_{shared} = 1 + .03*[40 + .04*180] = 2.416$

Benchmark B

$AMAT_{private} = 1 + .03*[5 + .02*360] = 1.37$

$AMAT_{shared} = 1 + .03*[40 + .01*180] = 2.25$

Private cache is superior for both benchmarks.

5.26.3

	Shared L2	Private L2
Single threaded	No advantage. No disadvantage.	No advantage. No disadvantage.
Multi-threaded	Shared caches can perform better for workloads where threads are tightly coupled and frequently share data. No disadvantage.	Threads often have private working sets, and using a private L2 prevents cache contamination and conflict misses between threads.
Multiprogrammed	No advantage except in rare cases where processes communicate. The disadvantage is higher cache latency.	Caches are kept private, isolating data between processes. This works especially well if the OS attempts to assign the same CPU to each process.

Having private L2 caches with a shared L3 cache is an effective compromise for many workloads, and this is the scheme used by many modern processors.

5.26.4 A non-blocking shared L2 cache would reduce the latency of the L2 cache by allowing hits for one CPU to be serviced while a miss is serviced for another CPU, or allow for misses from both CPUs to be serviced simultaneously. A non-blocking private L2 would reduce latency assuming that multiple memory instructions can be executed concurrently.

5.26.5 Four times.

5.26.6 Additional DRAM bandwidth, dynamic memory schedulers, multi-banked memory systems, higher cache associativity, and additional levels of cache.

5.27

5.27.1 `srcIP` and `refTime` fields. Two misses per entry.

5.27.2 Group the `srcIP` and `refTime` fields into a separate array. (I.e., create two parallel arrays. One with `srcIP` and `refTime`, and the other with the remaining fields.)

5.27.3 `peak_hour (int status); // peak hours of a given status`

Group `srcIP, refTime` and `status` together.

5.28

5.28.1 Answers will vary depending on which data set is used.

Conflict misses do not occur in fully associative caches.

Compulsory (cold) misses are not affected by associativity.

Capacity miss rate is computed by subtracting the compulsory miss rate and the fully associative miss rate (compulsory + capacity misses) from the total miss rate. Conflict miss rate is computed by subtracting the cold and the newly computed capacity miss rate from the total miss rate.

The values reported are miss rate per instruction, as opposed to miss rate per memory instruction.

5.28.2 Answers will vary depending on which data set is used.

5.28.3 Answers will vary.

5.29

5.29.1 Shadow page table: (1) VM creates page table, hypervisor updates shadow table; (2) nothing; (3) hypervisor intercepts page fault, creates new mapping, and invalidates the old mapping in TLB; (4) VM notifies the hypervisor to invalidate the process's TLB entries. Nested page table: (1) VM creates new page table, hypervisor adds new mappings in PA to MA table. (2) Hardware walks both page tables to translate VA to MA; (3) VM and hypervisor update their page tables, hypervisor invalidates stale TLB entries; (4) same as shadow page table.

5.29.2 Native: 4; NPT: 24 (instructors can change the levels of page table)

Native: L; NPT: $L \times (L + 2)$.

5.29.3 Shadow page table: page fault rate.

NPT: TLB miss rate.

5.29.4 Shadow page table: 1.03

NPT: 1.04.

5.29.5 Combining multiple page table updates.

5.29.6 NPT caching (similar to TLB caching).

6

Solutions

6.1 There is no single right answer for this question. The purpose is to get students to think about parallelism present in their daily lives. The answer should have at least 10 activities identified.

6.1.1 Any reasonable answer is correct here.

6.1.2 Any reasonable answer is correct here.

6.1.3 Any reasonable answer is correct here.

6.1.4 The student is asked to quantify the savings due to parallelism. The answer should consider the amount of overlap provided through parallelism and should be less than or equal to (if no parallelism was possible) the original time computed if each activity was carried out serially.

6.2

6.2.1 For this set of resources, we can pipeline the preparation. We assume that we do not have to reheat the oven for each cake.

Preheat Oven

Mix ingredients in bowl for Cake 1

Fill cake pan with contents of bowl and bake Cake 1. Mix ingredients for Cake 2 in bowl.

Finish baking Cake 1. Empty cake pan. Fill cake pan with bowl contents for Cake 2 and bake Cake 2. Mix ingredients in bowl for Cake 3.

Finish baking Cake 2. Empty cake pan. Fill cake pan with bowl contents for Cake 3 and bake Cake 3.

Finish baking Cake 3. Empty cake pan.

6.2.2 Now we have 3 bowls, 3 cake pans and 3 mixers. We will name them A, B, and C.

Preheat Oven

Mix incredients in bowl A for Cake 1

Fill cake pan A with contents of bowl A and bake for Cake 1. Mix ingredients for Cake 2 in bowl A.

Finish baking Cake 1. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 2. Mix ingredients in bowl A for Cake 3.

Finish baking Cake 2. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 3.

Finish baking Cake 3. Empty cake pan A.

The point here is that we cannot carry out any of these items in parallel because we either have one person doing the work, or we have limited capacity in our oven.

6.2.3 Each step can be done in parallel for each cake. The time to bake 1 cake, 2 cakes or 3 cakes is exactly the same.

6.2.4 The loop computation is equivalent to the steps involved to make one cake. Given that we have multiple processors (or ovens and cooks), we can execute instructions (or cook multiple cakes) in parallel. The instructions in the loop (or cooking steps) may have some dependencies on prior instructions (or cooking steps) in the loop body (cooking a single cake).

Data-level parallelism occurs when loop iterations are independent (i.e., no loop carried dependencies).

Task-level parallelism includes any instructions that can be computed on parallel execution units, similar to the independent operations involved in making multiple cakes.

6.3

6.3.1 While binary search has very good serial performance, it is difficult to parallelize without modifying the code. So part A asks to compute the speedup factor, but increasing X beyond 2 or 3 should have no benefits. While we can perform the comparison of low and high on one core, the computation for mid on a second core, and the comparison for A[mid] on a third core, without some restructuring or speculative execution, we will not obtain any speedup. The answer should include a graph, showing that no speedup is obtained after the values of 1, 2, or 3 (this value depends somewhat on the assumption made) for Y.

6.3.2 In this question, we suggest that we can increase the number of cores (to each of the number of array elements). Again, given the current code, we really cannot obtain any benefit from these extra cores. But if we create threads to compare the N elements to the value X and perform these in parallel, then we can get ideal speedup (Y times speedup), and the comparison can be completed in the amount of time to perform a single comparison.

6.4 This problem illustrates that some computations can be done in parallel if serial code is restructured. But, more importantly, we may want to provide for SIMD operations in our ISA, and allow for data-level parallelism when performing the same operation on multiple data items.

6.4.1 As shown below, each iteration of the loop requires 16 cycles. The loop runs 999 times. Thus, the total number of cycles is $16 \times 999 + 3 = 15984$.

```
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x10, 16
LOOP:   fld   f0, -16(x11)
        fld   f1, -1(x11)
        stall
        stall
        stall
        stall
        stall
        stall
        fadd.d f2, f0, f1
        stall
        stall
        stall
        stall
        fsd   f2, 0(x11)
        addi  x11, x11, 8
        ble   x11, x12, LOOP
```

6.4.2 The following code removes one stall per iteration:

a.
```
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x10, 16
LOOP:   fld   f0, -16(x11)
        fld   f1, -1(x11)
        stall
        stall
        stall
        stall
        stall
        stall
        fadd.d f2, f0, f1
        addi  x11, x11, 8
        stall
        stall
        stall
        fsd   f2, -8(x11)
        ble   x11, x12, LOOP
```

b. Thus, the new loop takes 15×999=14958 cycles.

6.4.3 Array elements `D[j]` and `D[j-1]` will have loop carried dependencies. The value loaded into `D0` during iteration `i` was produced during iteration `i-1`.

6.4.4

```
        li     x5, 8000
        add    x12, x10, x5
        fld    f0, 0(x11)
        fld    f1, 8(x11)
        addi   x11, x11, 8
        stall
        stall
        stall
        stall
        stall
LOOP:   fadd.d f2, f0, f1
        addi   x11, x11, 8
        fmv.d f0, f1
        fmv.d f1, f2
        stall
        fsd    f2, 0(x11)
        ble    x11, x12, LOOP
```

This loop takes seven cycles and runs 999 times. Thus, the total number of cycles is $7 \times 999 + 10 = 7003$.

6.4.5

```
        fld   f0, 0(x11)
        fld   f1, 8(x11)
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x11, 16
        stall
        stall
        stall
LOOP:   fadd.d f2, f0, f1
        stall
        stall
        stall
        stall
        fadd.d f0, f2, f1
        fsd   f2, 0(x11)
        stall
        stall
        stall
        fadd.d f1, f2, f0
        fsd   f0, 8(x11)
        addi  x11, x11, 24
        stall
```

```
stall
fsd   f1, -8(x11)
bne   x11, x12, LOOP
```

The unrolled loop takes 17 cycles, but runs only 333 times. Thus, the total number of cycles is $17 \times 333 + 10 = 5671$.

6.4.6 Include two copies of the loop: The unrolled loop and the original loop. Suppose you unrolled the loop U times. Run the unrolled loop until the number of iterations left is less than U. (In some sense your unrolled loop will be doing this: `for (i = 0; i + U < MAX; i+= U)`.) At this point, switch to the unrolled loop. (In some sense, your original loop will be doing this: `for (; i < MAX; i++)`.)

6.4.7 It is not possible to use message passing to improve performance—even if the message passing system has no latency. There is simply not enough work that can be done in parallel to benefit from using multiple CPUs. All the work that can be done in parallel can be scheduled between dependent floating point instructions.

6.5

6.5.1 This problem is again a divide and conquer problem, but utilizes recursion to produce a very compact piece of code. In part A the student is asked to compute the speedup when the number of cores is small. When forming the lists, we spawn a thread for the computation of left in the MergeSort code, and spawn a thread for the computation of the right. If we consider this recursively, for m initial elements in the array, we can utilize $1 + 2 + 4 + 8 + 16 + \ldots$. $\log_2$ (m) processors to obtain speedup.

6.5.2 In this question, $\log_2$ (m) is the largest value of Y for which we can obtain any speedup without restructuring. But if we had m cores, we could perform sorting using a very different algorithm. For instance, if we have greater than m/2 cores, we can compare all pairs of data elements, swap the elements if the left element is greater than the right element, and then repeat this step m times. So this is one possible answer for the question. It is known as parallel comparison sort. Various comparison sort algorithms include odd-even sort and cocktail sort.

6.6

6.6.1 This problem presents an "embarrassingly parallel" computation and asks the student to find the speedup obtained on a four-core system. The computations involved are: $(m \times p \times n)$ multiplications and $(m \times p \times (n-1))$ additions. The multiplications and additions associated with a single element in C are dependent (we cannot start summing up the results of the multiplications for an element until two products are available). So in this question, the speedup should be very close to 4.

6.6.2 This question asks about how speedup is affected due to cache misses caused by the four cores all working on different matrix elements that map to the same cache line. Each update would incur the cost of a cache miss, and so will reduce the speedup obtained by a factor of 3 times the cost of servicing a cache miss.

6.6.3 In this question, we are asked how to fix this problem. The easiest way to solve the false sharing problem is to compute the elements in C by traversing the matrix across columns instead of rows (i.e., using index-j instead of index-i). These elements will be mapped to different cache lines. Then we just need to make sure we process the matrix index that is computed (i, j) and (i + 1, j) on the same core. This will eliminate false sharing.

6.7

6.7.1 $x = 2, y = 2, w = 1, z = 0$
$x = 2, y = 2, w = 3, z = 0$
$x = 2, y = 2, w = 5, z = 0$
$x = 2, y = 2, w = 1, z = 2$
$x = 2, y = 2, w = 3, z = 2$
$x = 2, y = 2, w = 5, z = 2$
$x = 2, y = 2, w = 1, z = 4$
$x = 2, y = 2, w = 3, z = 4$
$x = 3, y = 2, w = 5, z = 4$

6.7.2 We could set synchronization instructions after each operation so that all cores see the same value on all nodes.

6.8

6.8.1 If every philosopher simultaneously picks up the left fork, then there will be no right fork to pick up. This will lead to starvation.

6.8.2 The basic solution is that whenever a philosopher wants to eat, she checks both forks. If they are free, then she eats. Otherwise, she waits until a neighbor contacts her. Whenever a philosopher finishes eating, she checks to see if her neighbors want to eat and are waiting. If so, then she releases the fork to one of them and lets them eat. The difficulty is to first be able to obtain both forks without another philosopher interrupting the transition between checking and acquisition. We can implement this a number of ways, but a simple way is to accept requests for forks in a centralized queue, and give out forks based on the priority defined by being closest to the head of the queue. This provides both deadlock prevention and fairness.

6.8.3 There are a number or right answers here, but basically showing a case where the request of the head of the queue does not have the closest forks available, though there are forks available for other philosophers.

6.8.4 By periodically repeating the request, the request will move to the head of the queue. This only partially solves the problem unless you can guarantee that all philosophers eat for exactly the same amount of time, and can use this time to schedule the issuance of the repeated request.

6.9

6.9.1

Core 1	Core 2
A3	B1, B4
A1, A2	B1, B4
A1, A4	B2
A1	B3

6.9.2 Answer is same as 6.9.1.

6.9.3

FU1	FU2
A1	A2
A1	
A1	
B1	B2
B1	
A3	
A4	
B2	
B4	

6.9.4

FU1	FU2
A1	B1
A1	B1
A1	B2
A2	B3
A3	B4
A4	

6.10 This is an open-ended question.

6.10.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.10.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.11

6.11.1 The answer should include an RISC-V program that includes four different processes that will compute ¼ of the sums. Assuming that memory latency is not an issue, the program should get linear speed when run on the four processors (there is no communication necessary between threads). If

memory is being considered in the answer, then the array blocking should consider preserving spatial locality so that false sharing is not created.

6.11.2 Since this program is highly data parallel and there are no data dependencies, an 8 × speedup should be observed. In terms of instructions, the SIMD machine should have fewer instructions (though this will depend upon the SIMD extensions).

6.12 This is an open-ended question that could have many possible answers. The key is that the student learns about MISD and compares it to an SIMD machine.

6.12.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.12.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.13 This is an open-ended question that could have many answers. The key is that the students learn about warps.

6.13.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.14 This is an open-ended programming assignment. The code should be tested for correctness.

6.14.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.14.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

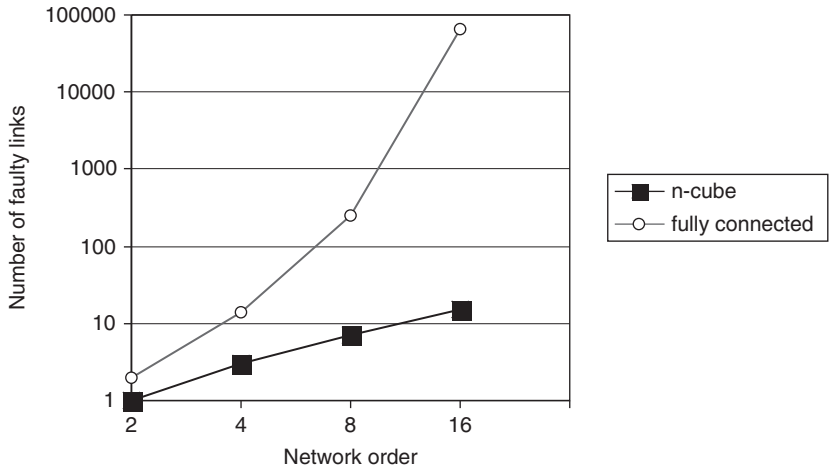
6.15 This question will require the students to research on the Internet both the AMD Fusion architecture and the Intel QuickPath technology. The key is that students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will change as the technology evolves.

6.15.1 There is no solution to this problem (it is a lab-based question-no need to change the solutions document).

6.16

6.16.1 For an n-cube of order N (2^N nodes), the interconnection network can sustain N − 1 broken links and still guarantee that there is a path to all nodes in the network.

6.16.2 The plot below shows the number of network links that can fail and still guarantee that the network is not disconnected.



6.17

6.17.1 Major differences between these suites include:

Whetstone—designed for floating point performance specifically

PARSEC—these workloads are focused on multithreaded programs.

6.17.2 Only the PARSEC benchmarks should be impacted by sharing and synchronization. This should not be a factor in Whetstone.

6.18

6.18.1 Any reasonable C program that performs the transformation should be accepted.

6.18.2 The storage space should be equal to (R+ R) times the size of a single precision floating point number+ (m+1) times the size of the index, where R is the number of non-zero elements and m is the number of rows. We will assume each floating-point number is 4 bytes, and each index is a short unsigned integer that is 2 bytes. For Matrix *X* this equals 111 bytes.

6.18.3 The answer should include results for both a brute-force and a computation using the Yale Sparse Matrix Format.

6.18.4 There are a number of more efficient formats, but their impact should be marginal for the small matrices used in this problem.

6.19

6.19.1 This question presents three different CPU models to consider when executing the following code:

```
if (X[i][j] > Y[i][j])
 count++;
```

6.19.2 There are a number of acceptable answers here, but they should consider the capabilities of each CPU and also its frequency. What follows is one possible answer:

Since X and Y are FP numbers, we should utilize the vector processor (CPU C) to issue two loads, eight matrix elements in parallel from A and eight matrix elements from B, into a single vector register and then perform a vector subtract. We would then issue two vector stores to put the result in memory.

Since the vector processor does not have comparison instructions, we would have CPU A perform two parallel conditional jumps based on floating point registers. We would increment two counts based on the conditional compare. Finally, we could just add the two counts for the entire matrix. We would not need to use core B.

6.20 This question looks at the amount of queuing that is occurring in the system given a maximum transaction processing rate, and the latency observed on average by a transaction. The latency includes both the service time (which is computed by the maximum rate) and the queue time.

6.20.1 So for a max transaction processing rate of 5000/sec, and we have four cores contributing, we would see an average latency of 0.8 ms if there was no queuing taking place. Thus, each core must have 1.25 transactions either executing or in some amount of completion on average.

So the answers are:

Latency	Max TP rate	Avg. # requests per core
1 ms	5000/sec	1.25
2 ms	5000/sec	2.5
1 ms	10,000/sec	2.5
2 ms	10,000/sec	5

6.20.2 We should be able to double the maximum transaction rate by doubling the number of cores.

6.20.3 The reason this does not happen is due to memory contention on the shared memory system.

B

APPENDIX

Graphics and Computing GPUs

John Nickolls
Director of Architecture
NVIDIA

David Kirk
Chief Scientist
NVIDIA

Imagination is more important than knowledge.

Albert Einstein
On Science, 1930s

B.1 Introduction

This appendix focuses on the GPU—the ubiquitous **graphics processing unit** in every PC, laptop, desktop computer, and workstation. In its most basic form, the GPU generates 2D and 3D graphics, images, and video that enable Window-based operating systems, graphical user interfaces, video games, visual imaging applications, and video. The modern GPU that we describe here is a highly parallel, highly multithreaded multiprocessor optimized for **visual computing**. To provide real-time visual interaction with computed objects via graphics, images, and video, the GPU has a unified graphics and computing architecture that serves as both a programmable graphics processor and a scalable parallel computing platform. PCs and game consoles combine a GPU with a CPU to form **heterogeneous systems**.

graphics processing unit (GPU) A processor optimized for 2D and 3D graphics, video, visual computing, and display.

visual computing A mix of graphics processing and computing that lets you visually interact with computed objects via graphics, images, and video.

heterogeneous system A system combining different processor types. A PC is a heterogeneous CPU–GPU system.

A Brief History of GPU Evolution

Fifteen years ago, there was no such thing as a GPU. Graphics on a PC were performed by a *video graphics array* (VGA) controller. A VGA controller was simply a memory controller and display generator connected to some DRAM. In the 1990s, semiconductor technology advanced sufficiently that more functions could be added to the VGA controller. By 1997, VGA controllers were beginning to incorporate some *three-dimensional* (3D) acceleration functions, including

hardware for triangle setup and rasterization (dicing triangles into individual pixels) and texture mapping and shading (applying "decals" or patterns to pixels and blending colors).

In 2000, the single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline and, therefore, deserved a new name beyond VGA controller. The term GPU was coined to denote that the graphics device had become a processor.

Over time, GPUs became more programmable, as programmable processors replaced fixed-function dedicated logic while maintaining the basic 3D graphics pipeline organization. In addition, computations became more precise over time, progressing from indexed arithmetic, to integer and fixed point, to single-precision floating-point, and recently to double-precision floating-point. GPUs have become massively parallel programmable processors with hundreds of cores and thousands of threads.

Recently, processor instructions and memory hardware were added to support general purpose programming languages, and a programming environment was created to allow GPUs to be programmed using familiar languages, including C and C++. This innovation makes a GPU a fully general-purpose, programmable, manycore processor, albeit still with some special benefits and limitations.

GPU Graphics Trends

GPUs and their associated drivers implement the OpenGL and DirectX models of graphics processing. OpenGL is an open standard for 3D graphics programming available for most computers. DirectX is a series of Microsoft multimedia programming interfaces, including Direct3D for 3D graphics. Since these **application programming interfaces (APIs)** have well-defined behavior, it is possible to build effective hardware acceleration of the graphics processing functions defined by the APIs. This is one of the reasons (in addition to increasing device density) why new GPUs are being developed every 12 to 18 months that double the performance of the previous generation on existing applications.

application programming interface (API) A set of function and data structure definitions providing an interface to a library of functions.

Frequent doubling of GPU performance enables new applications that were not previously possible. The intersection of graphics processing and parallel computing invites a new paradigm for graphics, known as visual computing. It replaces large sections of the traditional sequential hardware graphics pipeline model with programmable elements for geometry, vertex, and pixel programs. Visual computing in a modern GPU combines graphics processing and parallel computing in novel ways that permit new graphics algorithms to be implemented, and opens the door to entirely new parallel processing applications on pervasive high-performance GPUs.

Heterogeneous System

Although the GPU is arguably the most parallel and most powerful processor in a typical PC, it is certainly not the only processor. The CPU, now multicore and

soon to be manycore, is a complementary, primarily serial processor companion to the massively parallel manycore GPU. Together, these two types of processors comprise a heterogeneous multiprocessor system.

The best performance for many applications comes from using both the CPU and the GPU. This appendix will help you understand how and when to best split the work between these two increasingly parallel processors.

GPU Evolves into Scalable Parallel Processor

GPUs have evolved functionally from hardwired, limited capability VGA controllers to programmable parallel processors. This evolution has proceeded by changing the logical (API-based) graphics pipeline to incorporate programmable elements and also by making the underlying hardware pipeline stages less specialized and more programmable. Eventually, it made sense to merge disparate programmable pipeline elements into one unified array of many programmable processors.

In the GeForce 8-series generation of GPUs, the geometry, vertex, and pixel processing all run on the same type of processor. This unification allows for dramatic scalability. More programmable processor cores increase the total system throughput. Unifying the processors also delivers very effective load balancing, since any processing function can use the whole processor array. At the other end of the spectrum, a processor array can now be built with very few processors, since all of the functions can be run on the same processors.

Why CUDA and GPU Computing?

This uniform and scalable array of processors invites a new model of programming for the GPU. The large amount of floating-point processing power in the GPU processor array is very attractive for solving nongraphics problems. Given the large degree of parallelism and the range of scalability of the processor array for graphics applications, the programming model for more general computing must express the massive parallelism directly, but allow for scalable execution.

GPU computing Using a GPU for computing via a parallel programming language and API.

GPU computing is the term coined for using the GPU for computing via a parallel programming language and API, without using the traditional graphics API and graphics pipeline model. This is in contrast to the earlier **General Purpose computation on GPU (GPGPU)** approach, which involves programming the GPU using a graphics API and graphics pipeline to perform nongraphics tasks.

GPGPU Using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.

Compute Unifed Device Architecture (CUDA) is a scalable parallel programming model and software platform for the GPU and other parallel processors that allows the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. The CUDA programming model has an SPMD (single-program multiple data) software style, in which a programmer writes a program for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU. In fact, CUDA also provides a facility for programming multiple CPU cores as well, so CUDA is an environment for writing parallel programs for the entire heterogeneous computer system.

CUDA A scalable parallel programming model and language based on C/C++. It is a parallel programming platform for GPUs and multicore CPUs.

GPU Unifes Graphics and Computing

With the addition of CUDA and GPU computing to the capabilities of the GPU, it is now possible to use the GPU as both a graphics processor and a computing processor at the same time, and to combine these uses in visual computing applications. The underlying processor architecture of the GPU is exposed in two ways: first, as implementing the programmable graphics APIs, and second, as a massively parallel processor array programmable in C/C++ with CUDA.

Although the underlying processors of the GPU are unified, it is not necessary that all of the SPMD thread programs are the same. The GPU can run graphics shader programs for the graphics aspect of the GPU, processing geometry, vertices, and pixels, and also run thread programs in CUDA.

The GPU is truly a versatile multiprocessor architecture, supporting a variety of processing tasks. GPUs are excellent at graphics and visual computing as they were specifically designed for these applications. GPUs are also excellent at many general-purpose throughput applications that are "first cousins" of graphics, in that they perform a lot of parallel work, as well as having a lot of regular problem structure. In general, they are a good match to data-parallel problems (see Chapter 6), particularly large problems, but less so for less regular, smaller problems.

GPU Visual Computing Applications

Visual computing includes the traditional types of graphics applications plus many new applications. The original purview of a GPU was "anything with pixels," but it now includes many problems without pixels but with regular computation and/or data structure. GPUs are effective at 2D and 3D graphics, since that is the purpose for which they are designed. Failure to deliver this application performance would be fatal. 2D and 3D graphics use the GPU in its "graphics mode," accessing the processing power of the GPU through the graphics APIs, OpenGL™, and DirectX™. Games are built on the 3D graphics processing capability.

Beyond 2D and 3D graphics, image processing and video are important applications for GPUs. These can be implemented using the graphics APIs or as computational programs, using CUDA to program the GPU in computing mode. Using CUDA, image processing is simply another data-parallel array program. To the extent that the data access is regular and there is good locality, the program will be efficient. In practice, image processing is a very good application for GPUs. Video processing, especially encode and decode (compression and decompression according to some standard algorithms), is quite efficient.

The greatest opportunity for visual computing applications on GPUs is to "break the graphics pipeline." Early GPUs implemented only specific graphics APIs, albeit at very high performance. This was wonderful if the API supported the operations that you wanted to do. If not, the GPU could not accelerate your task, because early GPU functionality was immutable. Now, with the advent of GPU computing and CUDA, these GPUs can be programmed to implement a different virtual pipeline by simply writing a CUDA program to describe the computation and data flow that is desired. So, all applications are now possible, which will stimulate new visual computing approaches.

B.2 GPU System Architectures

In this section, we survey GPU system architectures in common use today. We discuss system configurations, GPU functions and services, standard programming interfaces, and a basic GPU internal architecture.

Heterogeneous CPU–GPU System Architecture

A heterogeneous computer system architecture using a GPU and a CPU can be described at a high level by two primary characteristics: first, how many functional subsystems and/or chips are used and what are their interconnection technologies and topology; and second, what memory subsystems are available to these functional subsystems. See Chapter 6 for background on the PC I/O systems and chip sets.

The Historical PC (circa 1990)

Figure B.2.1 shows a high-level block diagram of a legacy PC, circa 1990. The north bridge (see Chapter 6) contains high-bandwidth interfaces, connecting the CPU, memory, and PCI bus. The south bridge contains legacy interfaces and devices: ISA bus (audio, LAN), interrupt controller; DMA controller; time/counter. In this system, the display was driven by a simple framebuffer subsystem known

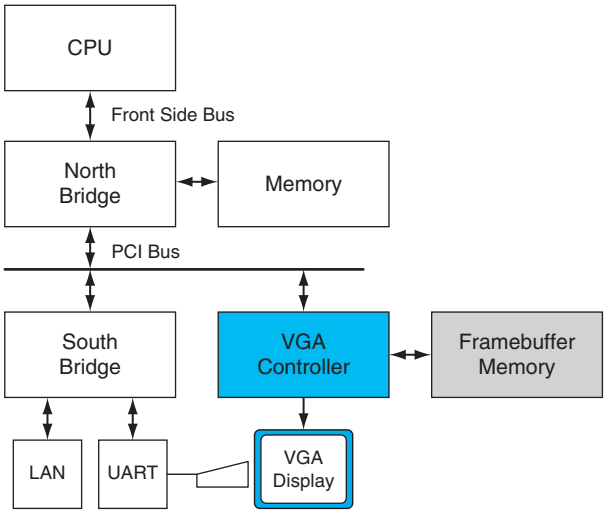


FIGURE B.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory.

PCI-Express (PCIe) A standard system I/O interconnect that uses point-to-point links. Links have a configurable number of lanes and bandwidth.

as a VGA (*video graphics array*) which was attached to the PCI bus. Graphics subsystems with built-in processing elements (GPUs) did not exist in the PC landscape of 1990.

Figure B.2.2 illustrates two confgurations in common use today. These are characterized by a separate GPU (discrete GPU) and CPU with respective memory subsystems. In Figure B.2.2a, with an Intel CPU, we see the GPU attached via a 16-lane **PCI-Express** 2.0 link to provide a peak 16 GB/s transfer rate (peak of 8 GB/s in each direction). Similarly, in Figure B.2.2b, with an AMD CPU, the GPU

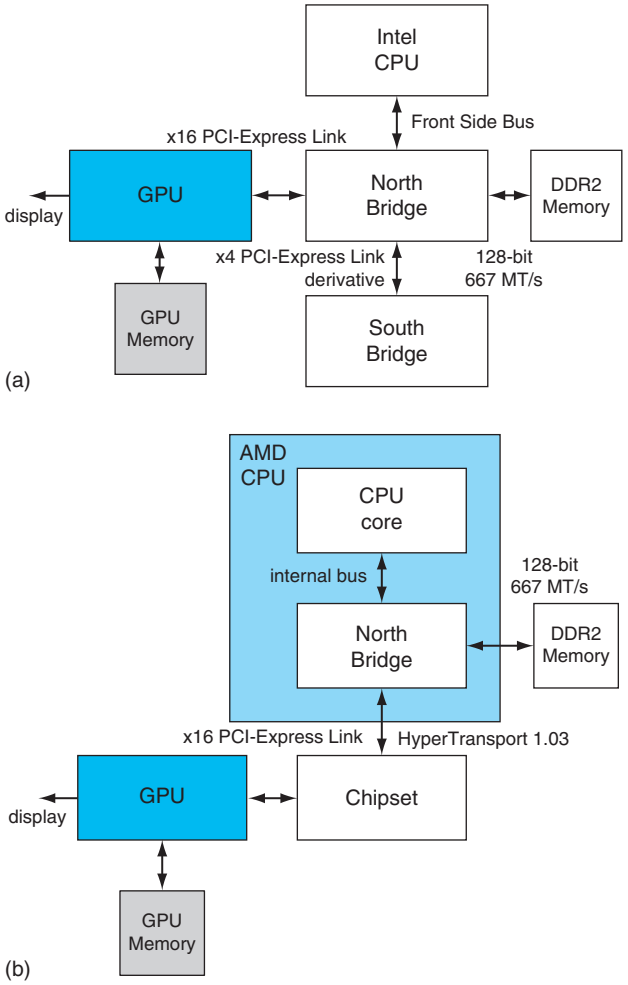


FIGURE B.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure.

is attached to the chipset, also via PCI-Express with the same available bandwidth. In both cases, the GPUs and CPUs may access each other's memory, albeit with less available bandwidth than their access to the more directly attached memories. In the case of the AMD system, the north bridge or memory controller is integrated into the same die as the CPU.

A low-cost variation on these systems, a **unified memory architecture (UMA)** system, uses only CPU system memory, omitting GPU memory from the system. These systems have relatively low-performance GPUs, since their achieved performance is limited by the available system memory bandwidth and increased latency of memory access, whereas dedicated GPU memory provides high bandwidth and low latency.

unified memory architecture (UMA) A system architecture in which the CPU and GPU share a common system memory.

A high-performance system variation uses multiple attached GPUs, typically two to four working in parallel, with their displays daisy-chained. An example is the NVIDIA SLI (scalable link interconnect) multi-GPU system, designed for high-performance gaming and workstations.

The next system category integrates the GPU with the north bridge (Intel) or chipset (AMD) with and without dedicated graphics memory.

Chapter 5 explains how caches maintain coherence in a shared address space. With CPUs and GPUs, there are multiple address spaces. GPUs can access their own physical local memory and the CPU system's physical memory using virtual addresses that are translated by an MMU on the GPU. The operating system kernel manages the GPU's page tables. A system physical page can be accessed using either coherent or noncoherent PCI-Express transactions, determined by an attribute in the GPU's page table. The CPU can access GPU's local memory through an address range (also called aperture) in the PCI-Express address space.

Game Consoles

Console systems such as the Sony PlayStation 3 and the Microsoft Xbox 360 resemble the PC system architectures previously described. Console systems are designed to be shipped with identical performance and functionality over a lifespan that can last five years or more. During this time, a system may be reimplemented many times to exploit more advanced silicon manufacturing processes and thereby to provide constant capability at ever lower costs. Console systems do not need to have their subsystems expanded and upgraded the way PC systems do, so the major internal system buses tend to be customized rather than standardized.

GPU Interfaces and Drivers

In a PC today, GPUs are attached to a CPU via PCI-Express. Earlier generations used **AGP**. Graphics applications call OpenGL [Segal and Akeley, 2006] or Direct3D [Microsoft DirectX Specifcation] API functions that use the GPU as a coprocessor. The APIs send commands, programs, and data to the GPU via a graphics device driver optimized for the particular GPU.

AGP An extended version of the original PCI I/O bus, which provided up to eight times the bandwidth of the original PCI bus to a single card slot. Its primary purpose was to connect graphics subsystems into PC systems.

Graphics Logical Pipeline

The graphics logical pipeline is described in Section B.3. Figure B.2.3 illustrates the major processing stages, and highlights the important programmable stages (vertex, geometry, and pixel shader stages).

FIGURE B.2.3 Graphics logical pipeline. Programmable graphics shader stages are blue, and fixed-function blocks are white.

Mapping Graphics Pipeline to Unified GPU Processors

Figure B.2.4 shows how the logical pipeline comprising separate independent programmable stages is mapped onto a physical distributed array of processors.

Basic Unified GPU Architecture

Unified GPU architectures are based on a parallel array of many programmable processors. They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier GPUs which had separate processors dedicated to each processing type. The programmable processor array is tightly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. Although the fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by an area, cost, or power budget, we will focus on the programmable processors here.

Compared with multicore CPUs, manycore GPUs have a different architectural design point, one focused on executing many parallel threads efficiently on many

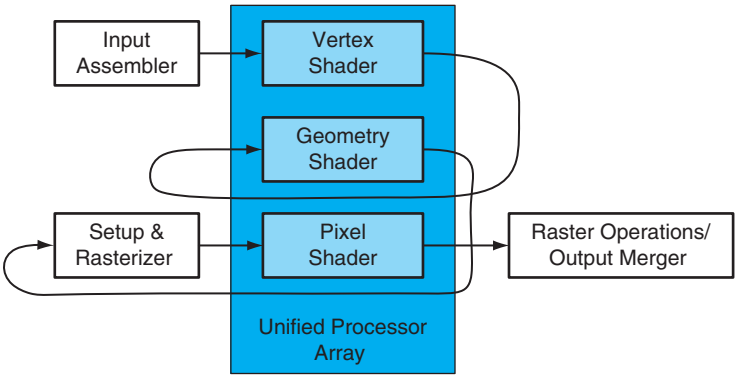


FIGURE B.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

processor cores. By using many simpler cores and optimizing for data-parallel behavior among groups of threads, more of the per-chip transistor budget is devoted to computation, and less to on-chip caches and overhead.

Processor Array

A unified GPU processor array contains many processor cores, typically organized into multithreaded multiprocessors. Figure B.2.5 shows a GPU with an array of 112 *streaming processor* (SP) cores, organized as 14 multithreaded *streaming multiprocessors* (SMs). Each SP core is highly multithreaded, managing 96 concurrent threads and their state in hardware. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. This is the basic Tesla architecture implemented by the NVIDIA GeForce 8800. It has a unified architecture in which the traditional graphics programs for vertex, geometry, and pixel shading run on the unified SMs and their SP cores, and computing programs run on the same processors.

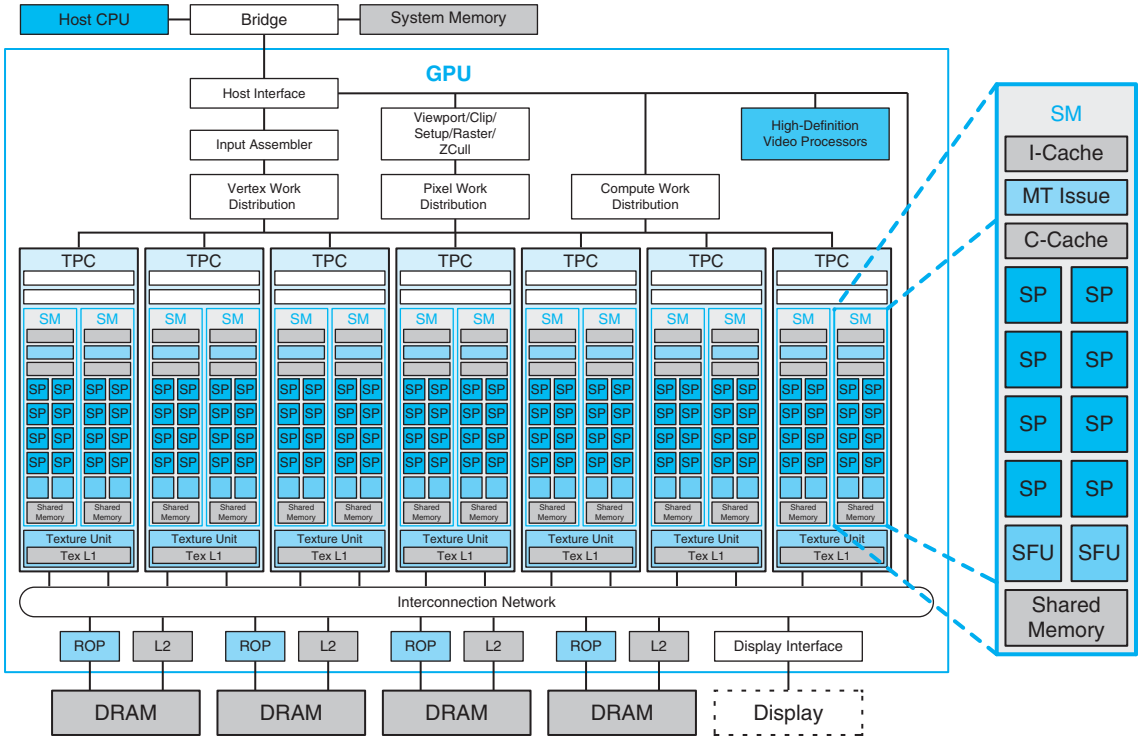


FIGURE B.2.5 Basic unified GPU architecture. Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

The processor array architecture is scalable to smaller and larger GPU configurations by scaling the number of multiprocessors and the number of memory partitions. Figure B.2.5 shows seven clusters of two SMs sharing a texture unit and a texture L1 cache. The texture unit delivers filtered results to the SM given a set of coordinates into a texture map. Because filter regions of support often overlap for successive texture requests, a small streaming L1 texture cache is effective to reduce the number of requests to the memory system. The processor array connects with *raster operation processors* (ROPs), L2 texture caches, external DRAM memories, and system memory via a GPU-wide interconnection network. The number of processors and number of memories can scale to design balanced GPU systems for different performance and market segments.

B.3 Programming GPUs

Programming multiprocessor GPUs is qualitatively different than programming other multiprocessors like multicore CPUs. GPUs provide two to three orders of magnitude more thread and data parallelism than CPUs, scaling to hundreds of processor cores and tens of thousands of concurrent threads. GPUs continue to increase their parallelism, doubling it about every 12 to 18 months, enabled by Moore's law [1965] of increasing integrated circuit density and by improving architectural efficiency. To span the wide price and performance range of different market segments, different GPU products implement widely varying numbers of processors and threads. Yet users expect games, graphics, imaging, and computing applications to work on any GPU, regardless of how many parallel threads it executes or how many parallel processor cores it has, and they expect more expensive GPUs (with more threads and cores) to run applications faster. As a result, GPU programming models and application programs are designed to scale transparently to a wide range of parallelism.

The driving force behind the large number of parallel threads and cores in a GPU is real-time graphics performance—the need to render complex 3D scenes with high resolution at interactive frame rates, at least 60 frames per second. Correspondingly, the scalable programming models of graphics shading languages such as Cg (C for graphics) and HLSL (*high-level shading language*) are designed to exploit large degrees of parallelism via many independent parallel threads and to scale to any number of processor cores. The CUDA scalable parallel programming model similarly enables general parallel computing applications to leverage large numbers of parallel threads and scale to any number of parallel processor cores, transparently to the application.

In these scalable programming models, the programmer writes code for a single thread, and the GPU runs myriad thread instances in parallel. Programs thus scale transparently over a wide range of hardware parallelism. This simple paradigm arose from graphics APIs and shading languages that describe how to shade one

vertex or one pixel. It has remained an effective paradigm as GPUs have rapidly increased their parallelism and performance since the late 1990s.

This section briefly describes programming GPUs for real-time graphics applications using graphics APIs and programming languages. It then describes programming GPUs for visual computing and general parallel computing applications using the C language and the CUDA programming model.

Programming Real-Time Graphics

APIs have played an important role in the rapid, successful development of GPUs and processors. There are two primary standard graphics APIs: **OpenGL** and **Direct3D**, one of the Microsoft DirectX multimedia programming interfaces. OpenGL, an open standard, was originally proposed and defined by Silicon Graphics Incorporated. The ongoing development and extension of the OpenGL standard [Segal and Akeley, 2006; Kessenich, 2006] is managed by Khronos, an industry consortium. Direct3D [Blythe, 2006], a de facto standard, is defined and evolved forward by Microsoft and partners. OpenGL and Direct3D are similarly structured, and continue to evolve rapidly with GPU hardware advances. They define a logical graphics processing pipeline that is mapped onto the GPU hardware and processors, along with programming models and languages for the programmable pipeline stages.

OpenGL An open-standard graphics API.

Direct3D A graphics API defined by Microsoft and partners.

Logical Graphics Pipeline

Figure B.3.1 illustrates the Direct3D 10 logical graphics pipeline. OpenGL has a similar graphics pipeline structure. The API and logical pipeline provide a streaming dataflow infrastructure and plumbing for the programmable shader stages, shown in blue. The 3D application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons. The input assembler collects vertices and primitives. The vertex shader program executes per-vertex processing,

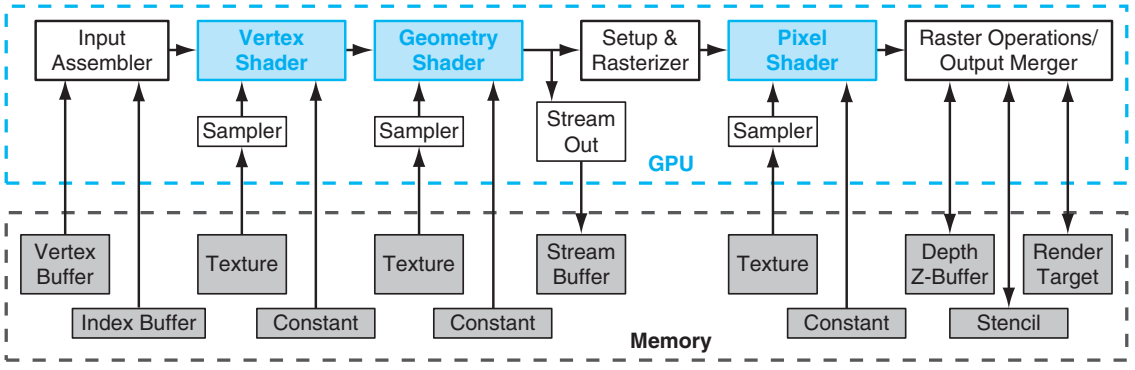


FIGURE B.3.1 Direct3D 10 graphics pipeline. Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are gray. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion.

including transforming the vertex 3D position into a screen position and lighting the vertex to determine its color. The geometry shader program executes per-primitive processing and can add or drop primitives. The setup and rasterizer unit generates pixel fragments (fragments are potential contributions to pixels) that are covered by a geometric primitive. The pixel shader program performs per-fragment processing, including interpolating per-fragment parameters, texturing, and coloring. Pixel shaders make extensive use of sampled and filtered lookups into large 1D, 2D, or 3D arrays called **textures**, using interpolated floating-point coordinates. Shaders use texture accesses for maps, functions, decals, images, and data. The raster operations processing (or output merger) stage performs Z-buffer depth testing and stencil testing, which may discard a hidden pixel fragment or replace the pixel's depth with the fragment's depth, and performs a color blending operation that combines the fragment color with the pixel color and writes the pixel with the blended color.

texture A 1D, 2D, or 3D array that supports sampled and filtered lookups with interpolated coordinates.

The graphics API and graphics pipeline provide input, output, memory objects, and infrastructure for the shader programs that process each vertex, primitive, and pixel fragment.

Graphics Shader Programs

Real-time graphics applications use many different **shader** programs to model how light interacts with different materials and to render complex lighting and shadows. **Shading languages** are based on a dataflow or streaming programming model that corresponds with the logical graphics pipeline. Vertex shader programs map the position of triangle vertices onto the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point *red, green, blue, alpha* (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. Shaders (and GPUs) use floating-point arithmetic for all pixel color calculations to eliminate visible artifacts while computing the extreme range of pixel contribution values encountered while rendering scenes with complex lighting, shadows, and high dynamic range. For all three types of graphics shaders, many program instances can be run in parallel, as independent parallel threads, because each works on independent data, produces independent results, and has no side effects. Independent vertices, primitives, and pixels further enable the same graphics program to run on differently sized GPUs that process different numbers of vertices, primitives, and pixels in parallel. Graphics programs thus scale transparently to GPUs with different amounts of parallelism and performance.

shader A program that operates on graphics data such as a vertex or a pixel fragment.

shading language A graphics rendering language, usually having a dataflow or streaming programming model.

Users program all three logical graphics threads with a common targeted high-level language. HLSL (high-level shading language) and Cg (C for graphics) are commonly used. They have C-like syntax and a rich set of library functions for matrix operations, trigonometry, interpolation, and texture access and filtering, but are far from general computing languages: they currently lack general memory

access, pointers, file I/O, and recursion. HLSL and Cg assume that programs live within a logical graphics pipeline, and thus I/O is implicit. For example, a pixel fragment shader may expect the geometric normal and multiple texture coordinates to have been interpolated from vertex values by upstream fixed-function stages and can simply assign a value to the COLOR output parameter to pass it downstream to be blended with a pixel at an implied (x, y) position.

The GPU hardware creates a new independent thread to execute a vertex, geometry, or pixel shader program for every vertex, every primitive, and every pixel fragment. In video games, the bulk of threads execute pixel shader programs, as there are typically 10 to 20 times more pixel fragments than vertices, and complex lighting and shadows require even larger ratios of pixel to vertex shader threads. The graphics shader programming model drove the GPU architecture to efficiently execute thousands of independent fine-grained threads on many parallel processor cores.

Pixel Shader Example

Consider the following Cg pixel shader program that implements the "environment mapping" rendering technique. For each pixel thread, this shader is passed five parameters, including 2D floating-point texture image coordinates needed to sample the surface color, and a 3D floating-point vector giving the refection of the view direction off the surface. The other three "uniform" parameters do not vary from one pixel instance (thread) to the next. The shader looks up color in two texture images: a 2D texture access for the surface color, and a 3D texture access into a cube map (six images corresponding to the faces of a cube) to obtain the external world color corresponding to the refection direction. Then the final four-component (red, green, blue, alpha) floating-point color is computed using a weighted average called a "lerp" or linear interpolation function.

```
void refection(
   float2               texCoord       : TEXCOORD0,
   float3               refection_dir  : TEXCOORD1,
   out float4           color          : COLOR,
   uniform float        shiny,
   uniform sampler2D    surfaceMap,
   uniform samplerCUBE  envMap)
{
// Fetch the surface color from a texture
   float4 surfaceColor = tex2D(surfaceMap, texCoord);

// Fetch reflected color by sampling a cube map
   float4 reflectedColor = texCUBE(environmentMap, refection_dir);

// Output is weighted average of the two colors
   color = lerp(surfaceColor, refectedColor, shiny);
}
```

Although this shader program is only three lines long, it activates a lot of GPU hardware. For each texture fetch, the GPU texture subsystem makes multiple memory accesses to sample image colors in the vicinity of the sampling coordinates, and then interpolates the final result with floating-point filtering arithmetic. The multithreaded GPU executes thousands of these lightweight Cg pixel shader threads in parallel, deeply interleaving them to hide texture fetch and memory latency.

Cg focuses the programmer's view to a single vertex or primitive or pixel, which the GPU implements as a single thread; the shader program transparently scales to exploit thread parallelism on the available processors. Being application-specific, Cg provides a rich set of useful data types, library functions, and language constructs to express diverse rendering techniques.

Figure B.3.2 shows skin rendered by a fragment pixel shader. Real skin appears quite different from flesh-color paint because light bounces around a lot before re-emerging. In this complex shader, three separate skin layers, each with unique subsurface scattering behavior, are modeled to give the skin a visual depth and translucency. Scattering can be modeled by a blurring convolution in a fattened "texture" space, with red being blurred more than green, and blue blurred less. The compiled Cg shader executes 1400 instructions to compute the color of one skin pixel.

FIGURE B.3.2 GPU-rendered image. To give the skin visual depth and translucency, the pixel shader program models three separate skin layers, each with unique subsurface scattering behavior. It executes 1400 instructions to render the red, green, blue, and alpha color components of each skin pixel fragment.

As GPUs have evolved superior floating-point performance and very high streaming memory bandwidth for real-time graphics, they have attracted highly parallel applications beyond traditional graphics. At first, access to this power was available only by couching an application as a graphics-rendering algorithm, but this GPGPU approach was often awkward and limiting. More recently, the CUDA programming model has provided a far easier way to exploit the scalable high-performance floating-point and memory bandwidth of GPUs with the C programming language.

Programming Parallel Computing Applications

CUDA, Brook, and CAL are programming interfaces for GPUs that are focused on data parallel computation rather than on graphics. CAL (*Compute Abstraction Layer*) is a low-level assembler language interface for AMD GPUs. Brook is a streaming language adapted for GPUs by Buck et al. [2004]. CUDA, developed by NVIDIA [2007], is an extension to the C and C+ + languages for scalable parallel programming of manycore GPUs and multicore CPUs. The CUDA programming model is described below, adapted from an article by Nickolls et al. [2008].

With the new model the GPU excels in data parallel and throughput computing, executing high-performance computing applications as well as graphics applications.

Data Parallel Problem Decomposition

To map large computing problems effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, such that the result blocks can be computed independently in parallel, and the elements within each block are computed in parallel. Figure B.3.3 shows a decomposition of a result data array into a 3×2 grid of blocks, where each block is further decomposed into a 5×3 array of elements. The two-level parallel decomposition maps naturally to the GPU architecture: parallel multiprocessors compute result blocks, and parallel threads compute result elements.

The programmer writes a program that computes a sequence of result data grids, partitioning each result grid into coarse-grained result blocks that can be computed independently in parallel. T0he program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads so that each computes one or more result elements.

Scalable Parallel Programming with CUDA

The CUDA scalable parallel programming model extends the C and C++ languages to exploit large degrees of parallelism for general applications on highly parallel multiprocessors, particularly GPUs. Early experience with CUDA shows that *many* sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have

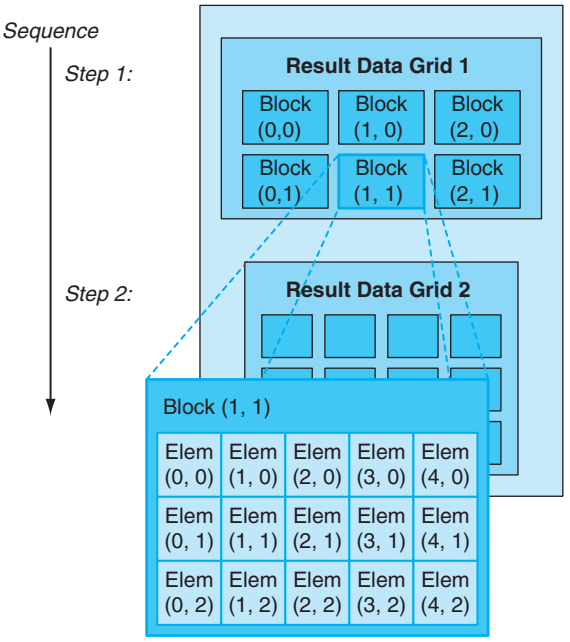


FIGURE B.3.3 Decomposing result data into a grid of blocks of elements to be computed in parallel.

rapidly developed scalable parallel programs for a wide range of applications, including seismic data processing, computational chemistry, linear algebra, sparse matrix solvers, sorting, searching, physics models, and visual computing. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the Tesla unified graphics and computing architecture (described in Sections B.4 and B.7) run CUDA C programs, and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared memory parallel processing architectures, including multicore CPUs.

CUDA provides three key abstractions—a *hierarchy of thread groups, shared memories*, and *barrier synchronization*—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel, and then into finer pieces that can be solved in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

The CUDA Paradigm

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel **kernels**, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of thread blocks and grids of thread blocks. A **thread block** is a set of concurrent threads that can cooperate among themselves through barrier synchronization and through shared access to a memory space private to the block. A **grid** is a set of thread blocks that may each be executed independently and thus may execute in parallel.

kernel A program or function for one thread, designed to be executed by many threads.

thread block A set of concurrent threads that execute the same thread program and may cooperate to compute a result.

grid A set of thread blocks that execute the same kernel program.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks comprising the grid. Each thread is given a unique *thread ID* number `threadIdx` within its thread block, numbered `0, 1, 2, …, blockDim-1`, and each thread block is given a unique *block ID* number `blockIdx` within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via `.x`, `.y`, and `.z` index fields.

As a very simple example of parallel programming, suppose that we are given two vectors x and y of n floating-point numbers each and that we wish to compute the result of $y = ax + y$ for some scalar value a. This is the so-called `SAXPY` kernel defined by the BLAS linear algebra library. Figure B.3.4 shows C code for performing this computation on both a serial processor and in parallel using CUDA.

The `__global__` declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function call syntax:

```
kernel<<<dimGrid, dimBlock>>>(… parameter list …);
```

where `dimGrid` and `dimBlock` are three-element vectors of type `dim3` that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to one.

In Figure B.3.4, we launch a grid of n threads that assigns one thread to each element of the vectors and puts 256 threads in each block. Each individual thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. Comparing the serial and parallel versions of this code, we see that they are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
     for(int i = 0; i<n; ++i)
          y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
     int i = blockIdx.x*blockDim.x + threadIdx.x;

     if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURE B.3.4 Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY (see Chapter 6). CUDA parallel threads replace the C serial loop—each thread computes the same result as one loop iteration. The parallel code computes *n* results with *n* threads organized in blocks of 256 threads.

Parallel execution and thread management is automatic. All thread creation, scheduling, and termination is handled for the programmer by the underlying system. Indeed, a Tesla architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a **synchronization barrier** by calling the `__syncthreads()` intrinsic. This guarantees that no thread in the block can proceed until all threads in the block have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by threads in the block before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

synchronization barrier Threads wait at a synchronization barrier until all threads in the thread block arrive at the barrier.

Since threads in a block may share memory and synchronize via barriers, they will reside together on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. The CUDA thread programming model virtualizes the processors and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. Virtualization

into threads and thread blocks allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. It also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks be able to execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using **atomic memory operations** on the global memory visible to all threads—by atomically incrementing queue pointers, for example. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary number of cores as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock. An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently, given sufficient hardware resources. Dependent grids execute sequentially, with an implicit interkernel barrier between them, thus guaranteeing that all blocks of the first grid complete before any block of the second, dependent grid begins.

atomic memory operation A memory read, modify, write operation sequence that completes without any intervening access.

Threads may access data from multiple memory spaces during their execution. Each thread has a private **local memory**. CUDA uses local memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a **shared memory**, visible to all threads of the block, which has the same lifetime as the block. Finally, all threads have access to the same **global memory**. Programs declare variables in shared and global memory with the `__shared__` and `__device__` type qualifers. On a Tesla architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

global memory Per-application memory shared by all threads.

shared memory Per-block memory shared by all threads of the block.

local memory Per-thread local memory private to the thread.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can therefore provide high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure B.3.5 shows diagrams of the nested levels of threads, thread blocks, and grids of thread blocks. It further shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as `cudaMalloc()` and `cudaFree()`. Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use `cudaMemcpy()` to copy data between the allocated space and the host system memory.

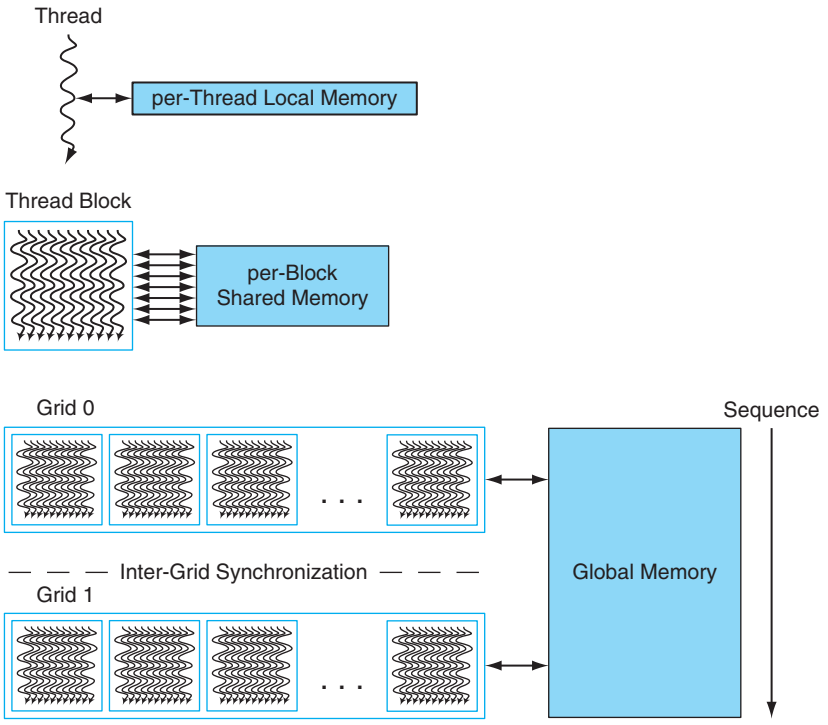


FIGURE B.3.5 Nested granularity levels—thread, thread block, and grid—have corresponding memory sharing levels—local, shared, and global. Per-thread local memory is private to the thread. Per-block shared memory is shared by all threads of the block. Per-application global memory is shared by all threads.

single-program multiple data (SPMD) A style of parallel programming model in which all threads execute the same program. SPMD threads typically coordinate with barrier synchronization.

The CUDA programming model is similar in style to the familiar **single-program multiple data (SPMD)** model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most realizations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads. Figure B.3.6 shows an example of an SPMD-like CUDA code sequence. It first instantiates `kernelF` on a 2D grid of 3 × 2 blocks where each 2D thread block consists of 5 × 3 threads. It then instantiates `kernelG` on a 1D grid of four 1D thread blocks with six threads each. Because `kernelG` depends on the results of `kernelF`, they are separated by an interkernel synchronization barrier.

The concurrent threads of a thread block express fine-grained data parallelism and thread parallelism. The independent thread blocks of a grid express coarse-grained data parallelism. Independent grids express coarse-grained task parallelism. A kernel is simply C code for one thread of the hierarchy.

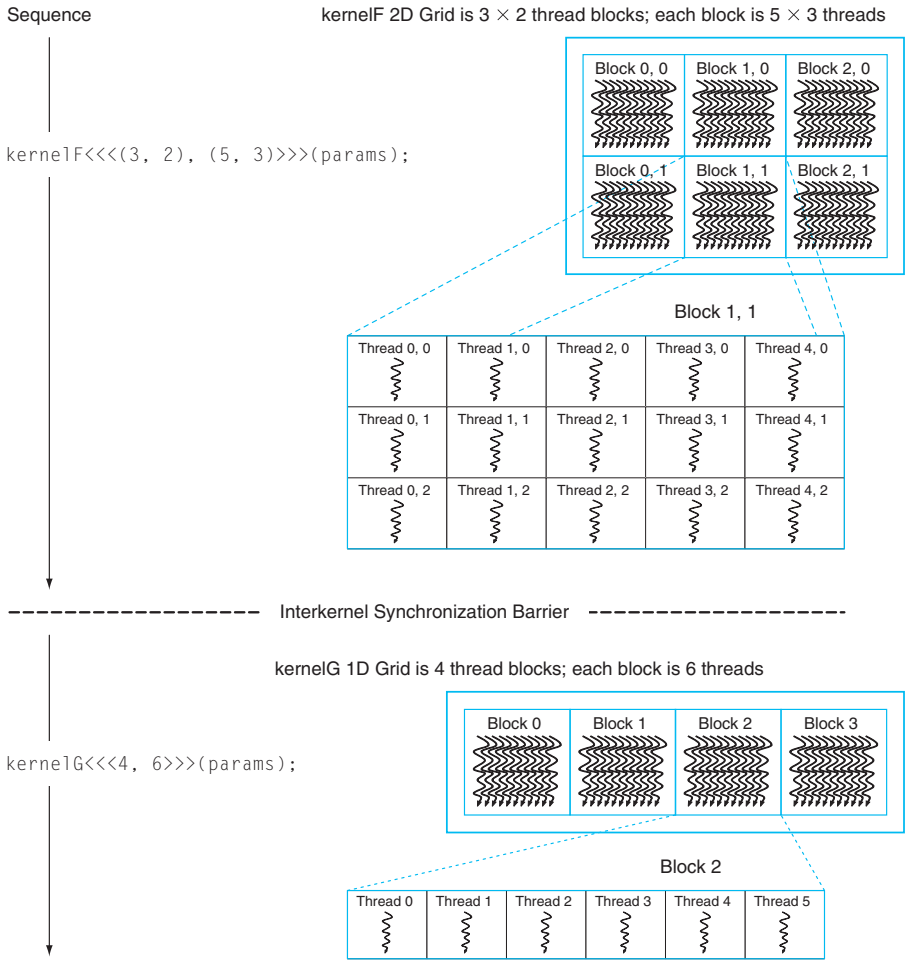


FIGURE B.3.6 Sequence of kernel *F* instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel *G* on a 1D grid of 1D thread blocks.

Restrictions

For efficiency, and to simplify its implementation, the CUDA programming model has some restrictions. Threads and thread blocks may only be created by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla GPU architecture implements *hardware* management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread block level but is difficult to express within a thread block because thread synchronization barriers operate on all the threads of the block. To enable CUDA programs to run on any number of processors, dependencies among thread blocks within the same kernel grid are not allowed—blocks must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks (although thread blocks may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example).

Recursive function calls are not currently allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel, because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block transfer engines and fast interconnects. Compute-intensive problems large enough to need a GPU performance boost amortize the overhead better than small problems.

Implications for Architecture

The parallel programming models for graphics and computing have driven GPU architecture to be different than CPU architecture. The key aspects of GPU programs driving GPU processor architecture are:

- *Extensive use of fine-grained data parallelism:* Shader programs describe how to process a single pixel or vertex, and CUDA programs describe how to compute an individual result.
- *Highly threaded programming model:* A shader thread program processes a single pixel or vertex, and a CUDA thread program may generate a single result. A GPU must create and execute millions of such thread programs per frame, at 60 frames per second.
- *Scalability:* A program must automatically increase its performance when provided with additional processors, without recompiling.
- *Intensive floating-point (or integer) computation*.
- *Support of high-throughput computations*.

B.4 Multithreaded Multiprocessor Architecture

To address different market segments, GPUs implement scalable numbers of multiprocessors—in fact, GPUs are multiprocessors composed of multiprocessors. Furthermore, each multiprocessor is highly multithreaded to execute many fine-grained vertex and pixel shader threads efficiently. A quality basic GPU has two to four multiprocessors, while a gaming enthusiast's GPU or computing platform has dozens of them. This section looks at the architecture of one such multithreaded multiprocessor, a simplified version of the NVIDIA Tesla *streaming multiprocessor* (SM) described in Section B.7.

Why use a multiprocessor, rather than several independent processors? The parallelism within each multiprocessor provides localized high performance and supports extensive multithreading for the fine-grained parallel programming models described in Section B.3. The individual threads of a thread block execute together within a multiprocessor to share data. The multithreaded multiprocessor design we describe here has eight scalar processor cores in a tightly coupled architecture, and executes up to 512 threads (the SM described in Section B.7 executes up to 768 threads). For area and power efficiency, the multiprocessor shares large complex units among the eight processor cores, including the instruction cache, the multithreaded instruction unit, and the shared memory RAM.

Massive Multithreading

GPU processors are highly multithreaded to achieve several goals:

- Cover the latency of memory loads and texture fetches from DRAM
- Support fine-grained parallel graphics shader programming models
- Support fine-grained parallel computing programming models
- Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- Simplify the parallel programming model to writing a serial program for one thread

Memory and texture fetch latency can require hundreds of processor clocks, because GPUs typically have small streaming caches rather than large working-set caches like CPUs. A fetch request generally requires a full DRAM access latency plus interconnect and buffering latency. Multithreading helps cover the latency with useful computing—while one thread is waiting for a load or texture fetch to complete, the processor can execute another thread. The fine-grained parallel programming models provide literally thousands of independent threads that can keep many processors busy despite the long memory latency seen by individual threads.

A graphics vertex or pixel shader program is a program for a single thread that processes a vertex or a pixel. Similarly, a CUDA program is a C program for a single thread that computes a result. Graphics and computing programs instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, each multiprocessor concurrently executes multiple different thread programs and different types of shader programs.

To support the independent vertex, primitive, and pixel programming model of graphics shading languages and the single-thread programming model of CUDA C/C++, each GPU thread has its own private registers, private per-thread memory, program counter, and thread execution state, and can execute an independent code path. To efficiently execute hundreds of concurrent lightweight threads, the GPU multiprocessor is hardware multithreaded—it manages and executes hundreds of concurrent threads in hardware without scheduling overhead. Concurrent threads within thread blocks can synchronize at a barrier with a single instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization efficiently support very fine-grained parallelism.

Multiprocessor Architecture

A unified graphics and computing multiprocessor executes vertex, geometry, and pixel fragment shader programs, and parallel computing programs. As Figure B.4.1 shows, the example multiprocessor consists of eight *scalar processor* (SP) cores each with a large multithreaded *register file* (RF), two *special function units* (SFUs), a multithreaded instruction unit, an instruction cache, a read-only constant cache, and a shared memory.

The 16 KB shared memory holds graphics data buffers and shared computing data. CUDA variables declared as `__shared__` reside in the shared memory. To map the logical graphics pipeline workload through the multiprocessor multiple times, as shown in Section B.2, vertex, geometry, and pixel threads have independent input and output buffers, and workloads arrive and depart independently of thread execution.

Each SP core contains scalar integer and floating-point arithmetic units that execute most instructions. The SP is hardware multithreaded, supporting up to 64 threads. Each pipelined SP core executes one scalar instruction per thread per clock, which ranges from 1.2 GHz to 1.6 GHz in different GPU products. Each SP core has a large RF of 1024 general-purpose 32-bit registers, partitioned among its assigned threads. Programs declare their register demand, typically 16 to 64 scalar 32-bit registers per thread. The SP can concurrently run many threads that use a few registers or fewer threads that use more registers. The compiler optimizes register allocation to balance the cost of spilling registers versus the cost of fewer threads. Pixel shader programs often use 16 or fewer registers, enabling each SP to run up to 64 pixel shader threads to cover long-latency texture fetches. Compiled CUDA programs often need 32 registers per thread, limiting each SP to 32 threads, which limits such a kernel program to 256 threads per thread block on this example multiprocessor, rather than its maximum of 512 threads.

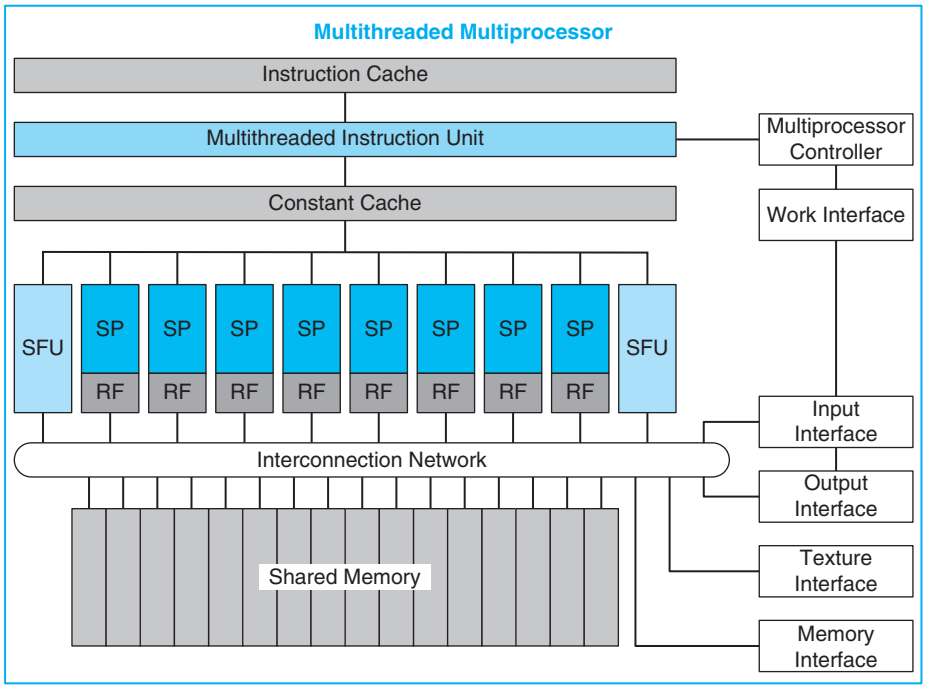


FIGURE B.4.1 Multithreaded multiprocessor with eight scalar processor (SP) cores. The eight SP cores each have a large multithreaded *register file* (RF) and share an instruction cache, multithreaded instruction issue unit, constant cache, two *special function units* (SFUs), interconnection network, and a multibank shared memory.

The pipelined SFUs execute thread instructions that compute special functions and interpolate pixel attributes from primitive vertex attributes. These instructions can execute concurrently with instructions on the SPs. The SFU is described later.

The multiprocessor executes texture fetch instructions on the texture unit via the texture interface, and uses the memory interface for external memory load, store, and atomic access instructions. These instructions can execute concurrently with instructions on the SPs. Shared memory access uses a low-latency interconnection network between the SP processors and the shared memory banks.

Single-Instruction Multiple-Thread (SIMT)

To manage and execute hundreds of threads running several different programs efficiently, the multiprocessor employs a **single-instruction multiple-thread (SIMT)** architecture. It creates, manages, schedules, and executes concurrent threads in groups of parallel threads called *warps*. The term **warp** originates from weaving, the first parallel thread technology. The photograph in Figure B.4.2 shows a warp of parallel threads emerging from a loom. This example multiprocessor uses a SIMT warp size of 32 threads, executing four threads in each of the eight SP cores over four

single-instruction multiple-thread (SIMT) A processor architecture that applies one instruction to multiple independent threads in parallel.

warp The set of parallel threads that execute the same instruction together in a SIMT architecture.

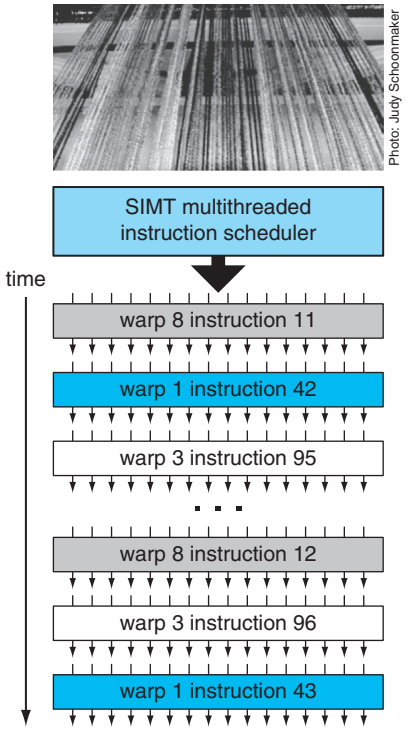


FIGURE B.4.2 SIMT multithreaded warp scheduling. The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time.

clocks. The Tesla SM multiprocessor described in Section B.7 also uses a warp size of 32 parallel threads, executing four threads per SP core for efficiency on plentiful pixel threads and computing threads. Thread blocks consist of one or more warps.

This example SIMT multiprocessor manages a pool of 16 warps, a total of 512 threads. Individual parallel threads composing a warp are the same type and start together at the same program address, but are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute its next instruction, and then issues that instruction to the active threads of that warp. A SIMT instruction is broadcast synchronously to the active parallel threads of a warp; individual threads may be inactive due to independent branching or predication. In this multiprocessor, each SP scalar processor core executes an instruction for four individual threads of a warp using four clocks, reflecting the 4:1 ratio of warp threads to cores.

SIMT processor architecture is akin to *single-instruction multiple data* (SIMD) design, which applies one instruction to multiple data lanes, but differs in that SIMT applies one instruction to multiple independent threads in parallel, not just

to multiple data lanes. An instruction for a SIMD processor controls a vector of multiple data lanes together, whereas an instruction for a SIMT processor controls an individual thread, and the SIMT instruction unit issues an instruction to a warp of independent parallel threads for efficiency. The SIMT processor finds data-level parallelism among threads at runtime, analogous to the way a superscalar processor finds instruction-level parallelism among instructions at runtime.

A SIMT processor realizes full efficiency and performance when all threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, execution serializes for each branch path taken, and when all paths complete, the threads converge to the same execution path. For equal length paths, a divergent if-else code block is 50% efficient. The multiprocessor uses a branch synchronization stack to manage independent threads that diverge and converge. Different warps execute independently at full speed regardless of whether they are executing common or disjoint code paths. As a result, SIMT GPUs are dramatically more efficient and flexible on branching code than earlier GPUs, as their warps are much narrower than the SIMD width of prior GPUs.

In contrast with SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for individual independent threads, as well as data-parallel code for many coordinated threads. For program correctness, the programmer can essentially ignore the SIMT execution attributes of warps; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional codes: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance.

SIMT Warp Execution and Divergence

The SIMT approach of scheduling independent warps is more flexible than the scheduling of previous GPU architectures. A warp comprises parallel threads of the same type: vertex, geometry, pixel, or compute. The basic unit of pixel fragment shader processing is the 2-by-2 pixel quad implemented as four pixel shader threads. The multiprocessor controller packs the pixel quads into a warp. It similarly groups vertices and primitives into warps, and packs computing threads into a warp. A thread block comprises one or more warps. The SIMT design shares the instruction fetch and issue unit efficiently across parallel threads of a warp, but requires a full warp of active threads to get full performance efficiency.

This unified multiprocessor schedules and executes multiple warp types concurrently, allowing it to concurrently execute vertex and pixel warps. Its warp scheduler operates at less than the processor clock rate, because there are four thread lanes per processor core. During each scheduling cycle, it selects a warp to execute a SIMT warp instruction, as shown in Figure B.4.2. An issued warp-instruction executes as four sets of eight threads over four processor cycles of throughput. The processor pipeline uses several clocks of latency to complete each instruction. If the number of active warps times the clocks per warp exceeds the pipeline

latency, the programmer can ignore the pipeline latency. For this multiprocessor, a round-robin schedule of eight warps has a period of 32 cycles between successive instructions for the same warp. If the program can keep 256 threads active per multiprocessor, instruction latencies up to 32 cycles can be hidden from an individual sequential thread. However, with few active warps, the processor pipeline depth becomes visible and may cause processors to stall.

A challenging design problem is implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types. The instruction scheduler must select a warp every four clocks to issue one instruction per clock per thread, equivalent to an IPC of 1.0 per processor core. Because warps are independent, the only dependences are among sequential instructions from the same warp. The scheduler uses a register dependency scoreboard to qualify warps whose active threads are ready to execute an instruction. It prioritizes all such ready warps and selects the highest priority one for issue. Prioritization must consider warp type, instruction type, and the desire to be fair to all active warps.

Managing Threads and Thread Blocks

The multiprocessor controller and instruction unit manage threads and thread blocks. The controller accepts work requests and input data and arbitrates access to shared resources, including the texture unit, memory access path, and I/O paths. For graphics workloads, it creates and manages three types of graphics threads concurrently: vertex, geometry, and pixel. Each of the graphics work types has independent input and output paths. It accumulates and packs each of these input work types into SIMT warps of parallel threads executing the same thread program. It allocates a free warp, allocates registers for the warp threads, and starts warp execution in the multiprocessor. Every program declares its per-thread register demand; the controller starts a warp only when it can allocate the requested register count for the warp threads. When all the threads of the warp exit, the controller unpacks the results and frees the warp registers and resources.

cooperative thread array (CTA) A set of concurrent threads that executes the same thread program and may cooperate to compute a result. A GPU CTA implements a CUDA thread block.

The controller creates **cooperative thread arrays (CTAs)** which implement CUDA thread blocks as one or more warps of parallel threads. It creates a CTA when it can create all CTA warps and allocate all CTA resources. In addition to threads and registers, a CTA requires allocating shared memory and barriers. The program declares the required capacities, and the controller waits until it can allocate those amounts before launching the CTA. Then it creates CTA warps at the warp scheduling rate, so that a CTA program starts executing immediately at full multiprocessor performance. The controller monitors when all threads of a CTA have exited, and frees the CTA shared resources and its warp resources.

Thread Instructions

The SP thread processors execute scalar instructions for individual threads, unlike earlier GPU vector instruction architectures, which executed four-component vector instructions for each vertex or pixel shader program. Vertex programs

generally compute (x, y, z, w) position vectors, while pixel shader programs compute (red, green, blue, alpha) color vectors. However, shader programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of a legacy GPU four-component vector architecture. In effect, the SIMT architecture parallelizes across 32 independent pixel threads, rather than parallelizing the four vector components within a pixel. CUDA C/C++ programs have predominantly scalar code per thread. Previous GPUs employed vector packing (e.g., combining subvectors of work to gain efficiency) but that complicated the scheduling hardware as well as the compiler. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

To support multiple GPUs with different binary microinstruction formats, high-level graphics and computing language compilers generate intermediate assembler-level instructions (e.g., Direct3D vector instructions or PTX scalar instructions), which are then optimized and translated to binary GPU microinstructions. The NVIDIA PTX (parallel thread execution) instruction set definition [2007] provides a stable target ISA for compilers, and provides compatibility over several generations of GPUs with evolving binary microinstruction-set architectures. The optimizer readily expands Direct3D vector instructions to multiple scalar binary microinstructions. PTX scalar instructions translate nearly one to one with scalar binary microinstructions, although some PTX instructions expand to multiple binary microinstructions, and multiple PTX instructions may fold into one binary microinstruction. Because the intermediate assembler-level instructions use virtual registers, the optimizer analyzes data dependencies and allocates real registers. The optimizer eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch diverge and converge points.

Instruction Set Architecture (ISA)

The thread ISA described here is a simplified version of the Tesla architecture PTX ISA, a register-based scalar instruction set comprising floating-point, integer, logical, conversion, special functions, flow control, memory access, and texture operations. Figure B.4.3 lists the basic PTX GPU thread instructions; see the NVIDIA PTX specification [2007] for details. The instruction format is:

```
opcode.type d, a, b, c;
```

where `d` is the destination operand, `a`, `b`, `c` are source operands, and `.type` is one of:

Type	.type Specifer
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating-point 16, 32, and 64 bits	.f16, .f32, .f64

Basic PTX GPU Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic *.type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a - b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.*type*	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.*type*	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 - a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.*cmp.type*	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric *.cmp* = eq, ne, lt, le, gt, ge; unordered *cmp* = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a;	move
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special Function	special *.type* = .f32 (some .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.*type*	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.*type*	sin.f32 d, a	d = sin(a);	sine
	cos.*type*	cos.f32 d, a	d = cos(a);	cosine
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic. *type* = .pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a, b	d = ~a;	one's complement
	cnot.*type*	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.*type*	shl.b32 d, a, b	d = a << b;	shift left
	shr.*type*	shr.s32 d, a, b	d = a >> b;	shift right
Memory Access	memory *.space* = .global, .shared, .local, .const; *.type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space.type*	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory *space*
	st.*space.type*	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory *space*
	tex.*nd.dtyp.btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.*spc.op.type*	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom *.op* = and, or, xor, add, min, max, exch, cas; *.spc* = .global; *.type* = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

FIGURE B.4.3 Basic PTX GPU thread instructions.

Source operands are scalar 32-bit or 64-bit values in registers, an immediate value, or a constant; predicate operands are 1-bit Boolean values. Destinations are registers, except for store to memory. Instructions are predicated by prefixing them with `@p` or `@!p`, where `p` is a predicate register. Memory and texture instructions transfer scalars or vectors of two to four components, up to 128 bits in total. PTX instructions specify the behavior of one thread.

The PTX arithmetic instructions operate on 32-bit and 64-bit floating-point, signed integer, and unsigned integer types. Recent GPUs support 64-bit double-precision floating-point; see Section B.6. On current GPUs, PTX 64-bit integer and logical instructions are translated to two or more binary microinstructions that perform 32-bit operations. The GPU special function instructions are limited to 32-bit floating-point. The thread control flow instructions are conditional `branch`, function `call` and `return`, thread `exit`, and `bar.sync` (barrier synchronization). The conditional branch instruction `@p bra target` uses a predicate register `p` (or `!p`) previously set by a compare and set predicate `setp` instruction to determine whether the thread takes the branch or not. Other instructions can also be predicated on a predicate register being true or false.

Memory Access Instructions

The `tex` instruction fetches and filters texture samples from 1D, 2D, and 3D texture arrays in memory via the texture subsystem. Texture fetches generally use interpolated floating-point coordinates to address a texture. Once a graphics pixel shader thread computes its pixel fragment color, the raster operations processor blends it with the pixel color at its assigned (x, y) pixel position and writes the final color to memory.

To support computing and C/C++ language needs, the Tesla PTX ISA implements memory load/store instructions. It uses integer byte addressing with register plus offset address arithmetic to facilitate conventional compiler code optimizations. Memory load/store instructions are common in processors, but are a significant new capability in the Tesla architecture GPUs, as prior GPUs provided only the texture and pixel accesses required by the graphics APIs.

For computing, the load/store instructions access three read/write memory spaces that implement the corresponding CUDA memory spaces in Section B.3:

- Local memory for per-thread private addressable temporary data (implemented in external DRAM)
- Shared memory for low-latency access to data shared by cooperating threads in the same CTA/thread block (implemented in on-chip SRAM)
- Global memory for large data sets shared by all threads of a computing application (implemented in external DRAM)

The memory load/store instructions `ld.global`, `st.global`, `ld.shared`, `st.shared`, `ld.local`, and `st.local` access the global, shared, and local memory spaces. Computing programs use the fast barrier synchronization instruction `bar.sync` to synchronize threads within a CTA/thread block that communicate with each other via shared and global memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread requests from the same SIMT warp together into a single memory block request when the addresses fall in the same block and meet alignment criteria. Coalescing memory requests provides a significant performance boost over separate requests from individual threads. The multiprocessor's large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

The latest Tesla architecture GPUs also provide efficient atomic memory operations on memory with the `atom.op.u32` instructions, including integer operations `add`, `min`, `max`, `and`, `or`, `xor`, `exchange`, and cas (compare-and-swap) operations, facilitating parallel reductions and parallel data structure management.

Barrier Synchronization for Thread Communication

Fast barrier synchronization permits CUDA programs to communicate frequently via shared memory and global memory by simply calling `__syncthreads();` as part of each interthread communication step. The synchronization intrinsic function generates a single `bar.sync` instruction. However, implementing fast barrier synchronization among up to 512 threads per CUDA thread block is a challenge.

Grouping threads into SIMT warps of 32 threads reduces the synchronization difficulty by a factor of 32. Threads wait at a barrier in the SIMT thread scheduler so they do not consume any processor cycles while waiting. When a thread executes a `bar.sync` instruction, it increments the barrier's thread arrival counter and the scheduler marks the thread as waiting at the barrier. Once all the CTA threads arrive, the barrier counter matches the expected terminal count, and the scheduler releases all the threads waiting at the barrier and resumes executing threads.

Streaming Processor (SP)

The multithreaded *streaming processor* (SP) core is the primary thread instruction processor in the multiprocessor. Its *register file* (RF) provides 1024 scalar 32-bit registers for up to 64 threads. It executes all the fundamental floating-point operations, including `add.f32`, `mul.f32`, `mad.f32` (floating multiply-add), `min.f32`, `max.f32`, and `setp.f32` (floating compare and set predicate). The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including *not-a-number* (NaN) and infinity values. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions shown in Figure B.4.3.

The floating-point `add` and `mul` operations employ IEEE round-to-nearest-even as the default rounding mode. The `mad.f32` floating-point multiply-add operation performs a multiplication with truncation, followed by an addition with round-to-nearest-even. The SP flushes input denormal operands to sign-preserved-zero. Results that underflow the target output exponent range are flushed to sign-preserved-zero after rounding.

Special Function Unit (SFU)

Certain thread instructions can execute on the SFUs, concurrently with other thread instructions executing on the SPs. The SFU implements the special function instructions of Figure B.4.3, which compute 32-bit floating-point approximations to reciprocal, reciprocal square root, and key transcendental functions. It also implements 32-bit floating-point planar attribute interpolation for pixel shaders, providing accurate interpolation of attributes such as color, depth, and texture coordinates.

Each pipelined SFU generates one 32-bit floating-point special function result per cycle; the two SFUs per multiprocessor execute special function instructions at a quarter the simple instruction rate of the eight SPs. The SFUs also execute the `mul.f32` multiply instruction concurrently with the eight SPs, increasing the peak computation rate up to 50% for threads with a suitable instruction mixture.

For functional evaluation, the Tesla architecture SFU employs quadratic interpolation based on enhanced minimax approximations for approximating the reciprocal, reciprocal square-root, $\log_2 x$, $2x$, and sin/cos functions. The accuracy of the function estimates ranges from 22 to 24 mantissa bits. See Section B.6 for more details on SFU arithmetic.

Comparing with Other Multiprocessors

Compared with SIMD vector architectures such as x86 SSE, the SIMT multiprocessor can execute individual threads independently, rather than always executing them together in synchronous groups. SIMT hardware finds data parallelism among independent threads, whereas SIMD hardware requires the software to express data parallelism explicitly in each vector instruction. A SIMT machine executes a warp of 32 threads synchronously when the threads take the same execution path, yet can execute each thread independently when they diverge. The advantage is significant because SIMT programs and instructions simply describe the behavior of a single independent thread, rather than a SIMD data vector of four or more data lanes. Yet the SIMT multiprocessor has SIMD-like efficiency, spreading the area and cost of one instruction unit across the 32 threads of a warp and across the eight streaming processor cores. SIMT provides the performance of SIMD together with the productivity of multithreading, avoiding the need to explicitly code SIMD vectors for edge conditions and partial divergence.

The SIMT multiprocessor imposes little overhead because it is hardware multithreaded with hardware barrier synchronization. That allows graphics shaders and CUDA threads to express very fine-grained parallelism. Graphics and CUDA programs use threads to express fine-grained data parallelism in a per-thread program, rather than forcing the programmer to express it as SIMD vector instructions. It is simpler and more productive to develop scalar single-thread code than vector code, and the SIMT multiprocessor executes the code with SIMD-like efficiency.

Coupling eight streaming processor cores together closely into a multiprocessor and then implementing a scalable number of such multiprocessors makes a two-level multiprocessor composed of multiprocessors. The CUDA programming model exploits the two-level hierarchy by providing individual threads for fine-grained parallel computations, and by providing grids of thread blocks for coarse-grained parallel operations. The same thread program can provide both fine-grained and coarse-grained operations. In contrast, CPUs with SIMD vector instructions must use two different programming models to provide fine-grained and coarse-grained operations: coarse-grained parallel threads on different cores, and SIMD vector instructions for fine-grained data parallelism.

Multithreaded Multiprocessor Conclusion

The example GPU multiprocessor based on the Tesla architecture is highly multithreaded, executing a total of up to 512 lightweight threads concurrently to support fine-grained pixel shaders and CUDA threads. It uses a variation on SIMD architecture and multithreading called SIMT (*single-instruction multiple-thread*) to efficiently broadcast one instruction to a warp of 32 parallel threads, while permitting each thread to branch and execute independently. Each thread executes its instruction stream on one of the eight *streaming processor* (SP) cores, which are multithreaded up to 64 threads.

The PTX ISA is a register-based load/store scalar ISA that describes the execution of a single thread. Because PTX instructions are optimized and translated to binary microinstructions for a specific GPU, the hardware instructions can evolve rapidly without disrupting compilers and software tools that generate PTX instructions.

B.5 Parallel Memory System

Outside of the GPU itself, the memory subsystem is the most important determiner of the performance of a graphics system. Graphics workloads demand very high transfer rates to and from memory. Pixel write and blend (read-modify-write) operations, depth buffer reads and writes, and texture map reads, as well as command and object vertex and attribute data reads, comprise the majority of memory traffic.

Modern GPUs are highly parallel, as shown in Figure B.2.5. For example, the GeForce 8800 can process 32 pixels per clock, at 600 MHz. Each pixel typically requires a color read and write and a depth read and write of a 4-byte pixel. Usually an average of two or three texels of four bytes each are read to generate the pixel's color. So for a typical case, there is a demand of 28 bytes times 32 pixels = 896 bytes per clock. Clearly the bandwidth demand on the memory system is enormous.

To supply these requirements, GPU memory systems have the following characteristics:

- They are wide, meaning there are a large number of pins to convey data between the GPU and its memory devices, and the memory array itself comprises many DRAM chips to provide the full total data bus width.
- They are fast, meaning aggressive signaling techniques are used to maximize the data rate (bits/second) per pin.
- GPUs seek to use every available cycle to transfer data to or from the memory array. To achieve this, GPUs specifically do not aim to minimize latency to the memory system. High throughput (utilization efficiency) and short latency are fundamentally in conflict.
- Compression techniques are used, both lossy, of which the programmer must be aware, and lossless, which is invisible to the application and opportunistic.
- Caches and work coalescing structures are used to reduce the amount of off-chip traffic needed and to ensure that cycles spent moving data are used as fully as possible.

DRAM Considerations

GPUs must take into account the unique characteristics of DRAM. DRAM chips are internally arranged as multiple (typically four to eight) banks, where each bank includes a power-of-2 number of rows (typically around 16,384), and each row contains a power-of-2 number of bits (typically 8192). DRAMs impose a variety of timing requirements on their controlling processor. For example, dozens of cycles are required to activate one row, but once activated, the bits within that row are randomly accessible with a new column address every four clocks. *Double-data rate* (DDR) synchronous DRAMs transfer data on both rising and falling edges of the interface clock (see Chapter 5). So a 1 GHz clocked DDR DRAM transfers data at 2 gigabits per second per data pin. Graphics DDR DRAMs usually have 32 bidirectional data pins, so eight bytes can be read or written from the DRAM per clock.

GPUs internally have a large number of generators of memory traffic. Different stages of the logical graphics pipeline each have their own request streams: command and vertex attribute fetch, shader texture fetch and load/store, and pixel depth and color read-write. At each logical stage, there are often multiple independent units to deliver the parallel throughput. These are each independent memory requestors. When viewed at the memory system, there is an enormous number of uncorrelated requests in flight. This is a natural mismatch to the reference pattern preferred by the DRAMs. A solution is for the GPU's memory controller to maintain separate heaps of traffic bound for different DRAM banks, and wait until enough traffic for

a particular DRAM row is pending before activating that row and transferring all the traffic at once. Note that accumulating pending requests, while good for DRAM row locality and thus efficient use of the data bus, leads to longer average latency as seen by the requestors whose requests spend time waiting for others. The design must take care that no particular request waits too long, otherwise some processing units can starve waiting for data and ultimately cause neighboring processors to become idle.

GPU memory subsystems are arranged as multiple *memory partitions*, each of which comprises a fully independent memory controller and one or two DRAM devices that are fully and exclusively owned by that partition. To achieve the best load balance and therefore approach the theoretical performance of n partitions, addresses are finely interleaved evenly across all memory partitions. The partition interleaving stride is typically a block of a few hundred bytes. The number of memory partitions is designed to balance the number of processors and other memory requesters.

Caches

GPU workloads typically have very large working sets—on the order of hundreds of megabytes to generate a single graphics frame. Unlike with CPUs, it is not practical to construct caches on chips large enough to hold anything close to the full working set of a graphics application. Whereas CPUs can assume very high cache hit rates (99.9% or more), GPUs experience hit rates closer to 90% and must therefore cope with many misses in flight. While a CPU can reasonably be designed to halt while waiting for a rare cache miss, a GPU needs to proceed with misses and hits intermingled. We call this a *streaming cache architecture*.

GPU caches must deliver very high-bandwidth to their clients. Consider the case of a texture cache. A typical texture unit may evaluate two bilinear interpolations for each of four pixels per clock cycle, and a GPU may have many such texture units all operating independently. Each bilinear interpolation requires four separate texels, and each texel might be a 64-bit value. Four 16-bit components are typical. Thus, total bandwidth is $2 \times 4 \times 4 \times 64 = 2048$ bits per clock. Each separate 64-bit texel is independently addressed, so the cache needs to handle 32 unique addresses per clock. This naturally favors a multibank and/or multiport arrangement of SRAM arrays.

MMU

Modern GPUs are capable of translating virtual addresses to physical addresses. On the GeForce 8800, all processing units generate memory addresses in a 40-bit virtual address space. For computing, load and store thread instructions use 32-bit byte addresses, which are extended to a 40-bit virtual address by adding a 40-bit offset. A memory management unit performs virtual to physical address

translation; hardware reads the page tables from local memory to respond to misses on behalf of a hierarchy of translation lookaside buffers spread out among the processors and rendering engines. In addition to physical page bits, GPU page table entries specify the compression algorithm for each page. Page sizes range from 4 to 128 kilobytes.

Memory Spaces

As introduced in Section B.3, CUDA exposes different memory spaces to allow the programmer to store data values in the most performance-optimal way. For the following discussion, NVIDIA Tesla architecture GPUs are assumed.

Global memory

Global memory is stored in external DRAM; it is not local to any one physical *streaming multiprocessor* (SM) because it is meant for communication among different CTAs (thread blocks) in different grids. In fact, the many CTAs that reference a location in global memory may not be executing in the GPU at the same time; by design, in CUDA a programmer does not know the relative order in which CTAs are executed. Because the address space is evenly distributed among all memory partitions, there must be a read/write path from any streaming multiprocessor to any DRAM partition.

Access to global memory by different threads (and different processors) is not guaranteed to have sequential consistency. Thread programs see a relaxed memory ordering model. Within a thread, the order of memory reads and writes to the same address is preserved, but the order of accesses to different addresses may not be preserved. Memory reads and writes requested by different threads are unordered. Within a CTA, the barrier synchronization instruction `bar.sync` can be used to obtain strict memory ordering among the threads of the CTA. The `membar` thread instruction provides a memory barrier/fence operation that commits prior memory accesses and makes them visible to other threads before proceeding. Threads can also use the atomic memory operations described in Section B.4 to coordinate work on memory they share.

Shared memory

Per-CTA shared memory is only visible to the threads that belong to that CTA, and shared memory only occupies storage from the time a CTA is created to the time it terminates. Shared memory can therefore reside on-chip. This approach has many benefits. First, shared memory traffc does not need to compete with limited off-chip bandwidth needed for global memory references. Second, it is practical to build very high-bandwidth memory structures on-chip to support the read/write demands of each streaming multiprocessor. In fact, the shared memory is closely coupled to the streaming multiprocessor.

Each streaming multiprocessor contains eight physical thread processors. During one shared memory clock cycle, each thread processor can process two threads' worth of instructions, so 16 threads' worth of shared memory requests must be handled in each clock. Because each thread can generate its own addresses, and the addresses are typically unique, the shared memory is built using 16 independently addressable SRAM banks. For common access patterns, 16 banks are sufficient to maintain throughput, but pathological cases are possible; for example, all 16 threads might happen to access a different address on one SRAM bank. It must be possible to route a request from any thread lane to any bank of SRAM, so a 16-by-16 interconnection network is required.

Local Memory

Per-thread local memory is private memory visible only to a single thread. Local memory is architecturally larger than the thread's register file, and a program can compute addresses into local memory. To support large allocations of local memory (recall the total allocation is the per-thread allocation times the number of active threads), local memory is allocated in external DRAM.

Although global and per-thread local memory reside off-chip, they are well-suited to being cached on-chip.

Constant Memory

Constant memory is read-only to a program running on the SM (it can be written via commands to the GPU). It is stored in external DRAM and cached in the SM. Because commonly most or all threads in a SIMT warp read from the same address in constant memory, a single address lookup per clock is sufficient. The constant cache is designed to broadcast scalar values to threads in each warp.

Texture Memory

Texture memory holds large read-only arrays of data. Textures for computing have the same attributes and capabilities as textures used with 3D graphics. Although textures are commonly two-dimensional images (2D arrays of pixel values), 1D (linear) and 3D (volume) textures are also available.

A compute program references a texture using a `tex` instruction. Operands include an identifier to name the texture, and one, two, or three coordinates based on the texture dimensionality. The floating-point coordinates include a fractional portion that specifies a sample location, often in-between texel locations. Noninteger coordinates invoke a bilinear weighted interpolation of the four closest values (for a 2D texture) before the result is returned to the program.

Texture fetches are cached in a streaming cache hierarchy designed to optimize throughput of texture fetches from thousands of concurrent threads. Some programs use texture fetches as a way to cache global memory.

Surfaces

Surface is a generic term for a one-dimensional, two-dimensional, or three-dimensional array of pixel values and an associated format. A variety of formats are defined; for example, a pixel may be defined as four 8-bit RGBA integer components, or four 16-bit floating-point components. A program kernel does not need to know the surface type. A `tex` instruction recasts its result values as floating-point, depending on the surface format.

Load/Store Access

Load/store instructions with integer byte addressing enable the writing and compiling of programs in conventional languages like C and C++. CUDA programs use load/store instructions to access memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread requests from the same warp together into a single memory block request when the addresses fall in the same block and meet alignment criteria. Coalescing individual small memory requests into large block requests provides a significant performance boost over separate requests. The large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

ROP

As shown in Figure B.2.5, NVIDIA Tesla architecture GPUs comprise a scalable *streaming processor array* (SPA), which performs all of the GPU's programmable calculations, and a scalable memory system, which comprises external DRAM control and fixed function *Raster Operation Processors* (ROPs) that perform color and depth framebuffer operations directly on memory. Each ROP unit is paired with a specific memory partition. ROP partitions are fed from the SMs via an interconnection network. Each ROP is responsible for depth and stencil tests and updates, as well as color blending. The ROP and memory controllers cooperate to implement lossless color and depth compression (up to 8:1) to reduce external bandwidth demand. ROP units also perform atomic operations on memory.

B.6 Floating-point Arithmetic

GPUs today perform most arithmetic operations in the programmable processor cores using IEEE 754-compatible single precision 32-bit floating-point operations (see Chapter 3). The fixed-point arithmetic of early GPUs was succeeded by 16-bit, 24-bit, and 32-bit floating-point, then IEEE 754-compatible 32-bit floating-point.

Some fixed-function logic within a GPU, such as texture-filtering hardware, continues to use proprietary numeric formats. Recent GPUs also provide IEEE 754-compatible double-precision 64-bit floating-point instructions.

Supported Formats

The IEEE 754 standard for floating-point arithmetic specifies basic and storage formats. GPUs use two of the basic formats for computation, 32-bit and 64-bit binary floating-point, commonly called single precision and double precision. The standard also specifies a 16-bit binary storage floating-point format, **half precision**. GPUs and the Cg shading language employ the narrow 16-bit half data format for efficient data storage and movement, while maintaining high dynamic range. GPUs perform many texture filtering and pixel blending computations at half precision within the texture filtering unit and the raster operations unit. The OpenEXR high dynamic-range image file format developed by Industrial Light and Magic [2003] uses the identical half format for color component values in computer imaging and motion picture applications.

half precision A 16-bit binary floating-point format, with 1 sign bit, 5-bit exponent, 10-bit fraction, and an implied integer bit.

Basic Arithmetic

Common single-precision floating-point operations in GPU programmable cores include addition, multiplication, **multiply-add**, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. Floating-point instructions often provide source operand modifiers for negation and absolute value.

multiply-add (MAD) A single floating-point instruction that performs a compound operation: multiplication followed by addition.

The floating-point addition and multiplication operations of most GPUs today are compatible with the IEEE 754 standard for single precision FP numbers, including *not-a-number* (NaN) and infinity values. The FP addition and multiplication operations use IEEE round-to-nearest-even as the default rounding mode. To increase floating-point instruction throughput, GPUs often use a compound multiply-add instruction (`mad`). The multiply-add operation performs FP multiplication with truncation, followed by FP addition with round-to-nearest-even. It provides two floating-point operations in one issuing cycle, without requiring the instruction scheduler to dispatch two separate instructions, but the computation is not fused and truncates the product before the addition. This makes it different from the fused multiply-add instruction discussed in Chapter 3 and later in this section. GPUs typically flush denormalized source operands to sign-preserved zero, and they flush results that underflow the target output exponent range to sign-preserved zero after rounding.

Specialized Arithmetic

GPUs provide hardware to accelerate special function computation, attribute interpolation, and texture filtering. Special function instructions include cosine,

sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root. Attribute interpolation instructions provide efficient generation of pixel attributes, derived from plane equation evaluation. The **special function unit (SFU)** introduced in Section B.4 computes special functions and interpolates planar attributes [Oberman and Siu, 2005].

special function unit (SFU) A hardware unit that computes special functions and interpolates planar attributes.

Several methods exist for evaluating special functions in hardware. It has been shown that quadratic interpolation based on Enhanced Minimax Approximations is a very efficient method for approximating functions in hardware, including reciprocal, reciprocal square-root, $\log_2 x$, 2^x, sin, and cos.

We can summarize the method of SFU quadratic interpolation. For a binary input operand X with n-bit significand, the significand is divided into two parts: X_u is the upper part containing m bits, and X_l is the lower part containing n-m bits. The upper m bits X_u are used to consult a set of three lookup tables to return three finite-word coefficients C_0, C_1, and C_2. Each function to be approximated requires a unique set of tables. These coefficients are used to approximate a given function f(X) in the range $X_u \le X < X_u + 2^{-m}$ by evaluating the expression:

$$f(X) = C_0 + C_1 X_1 + C_2 X_1^2$$

The accuracy of each of the function estimates ranges from 22 to 24 significand bits. Example function statistics are shown in Figure B.6.1.

The IEEE 754 standard specifies exact-rounding requirements for division and square root; however, for many GPU applications, exact compliance is not required. Rather, for those applications, higher computational throughput is more important than last-bit accuracy. For the SFU special functions, the CUDA math library provides both a full accuracy function and a fast function with the SFU instruction accuracy.

Another specialized arithmetic operation in a GPU is attribute interpolation. Key *attributes* are usually specified for vertices of primitives that make up a scene to be rendered. Example attributes are color, depth, and texture coordinates. These attributes must be interpolated in the (x,y) screen space as needed to determine the

Function	Input interval	Accuracy (good bits)	ULP* error	% exactly rounded	Monotonic
$1/x$	[1, 2)	24.02	0.98	87	Yes
$1/\text{sqrt}(x)$	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
sin/cos	[0, $\pi/2$)	22.47	N/A	N/A	No

*ULP: unit in the last place.**N/A: not applicable.

FIGURE B.6.1 Special function approximation statistics. For the NVIDIA GeForce 8800 *special function unit* (SFU).

values of the attributes at each pixel location. The value of a given attribute *U* in an (x, y) plane can be expressed using plane equations of the form:

$$U(x,y) = A_u x + B_u Y + C_u$$

where *A*, *B*, and *C* are interpolation parameters associated with each attribute *U*. The interpolation parameters *A*, *B*, and *C* are all represented as single-precision floating-point numbers.

Given the need for both a function evaluator and an attribute interpolator in a pixel shader processor, a single SFU that performs both functions for efficiency can be designed. Both functions use a sum of products operation to interpolate results, and the number of terms to be summed in both functions is very similar.

Texture Operations

Texture mapping and filtering is another key set of specialized floating-point arithmetic operations in a GPU. The operations used for texture mapping include:

1. Receive texture address (s, t) for the current screen pixel (x, y), where s and t are single-precision floating-point numbers.
2. Compute the level of detail to identify the correct texture **MIP-map** level.
3. Compute the trilinear interpolation fraction.
4. Scale texture address (s, t) for the selected MIP-map level.
5. Access memory and retrieve desired texels (texture elements).
6. Perform filtering operation on texels.

MIP-map A Latin phrase *multum in parvo*, or much in a small space. A MIP-map contains precalculated images of different resolutions, used to increase rendering speed and reduce artifacts.

Texture mapping requires a significant amount of floating-point computation for full-speed operation, much of which is done at 16-bit half precision. As an example, the GeForce 8800 Ultra delivers about 500 GFLOPS of proprietary format floating-point computation for texture mapping instructions, in addition to its conventional IEEE single-precision floating-point instructions. For more details on texture mapping and filtering, see Foley and van Dam [1995].

Performance

The floating-point addition and multiplication arithmetic hardware is fully pipelined, and latency is optimized to balance delay and area. While pipelined, the throughput of the special functions is less than the floating-point addition and multiplication operations. Quarter-speed throughput for the special functions is typical performance in modern GPUs, with one SFU shared by four SP cores. In contrast, CPUs typically have significantly lower throughput for similar functions, such as division and square root, albeit with more accurate results. The attribute interpolation hardware is typically fully pipelined to enable full-speed pixel shaders.

Double precision

Newer GPUs such as the Tesla T10P also support IEEE 754 64-bit double-precision operations in hardware. Standard floating-point arithmetic operations in double precision include addition, multiplication, and conversions between different floating-point and integer formats. The 2008 IEEE 754 floating-point standard includes specification for the *fused-multiply-add* (FMA) operation, as discussed in Chapter 3. The FMA operation performs a floating-point multiplication followed by an addition, with a single rounding. The fused multiplication and addition operations retain full accuracy in intermediate calculations. This behavior enables more accurate floating-point computations involving the accumulation of products, including dot products, matrix multiplication, and polynomial evaluation. The FMA instruction also enables efficient software implementations of exactly rounded division and square root, removing the need for a hardware division or square root unit.

A double-precision hardware FMA unit implements 64-bit addition, multiplication, conversions, and the FMA operation itself. The architecture of a

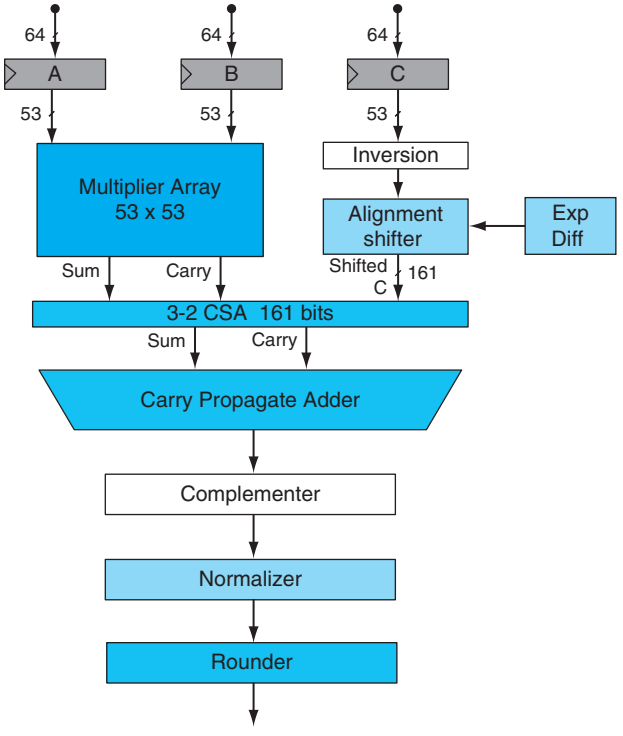


FIGURE B.6.2 Double-precision fused-multiply-add (FMA) unit. Hardware to implement floating-point A× B+ C for double precision.

double-precision FMA unit enables full-speed denormalized number support on both inputs and outputs. Figure B.6.2 shows a block diagram of an FMA unit.

As shown in Figure B.6.2, the significands of A and B are multiplied to form a 106-bit product, with the results left in carry-save form. In parallel, the 53-bit addend C is conditionally inverted and aligned to the 106-bit product. The sum and carry results of the 106-bit product are summed with the aligned addend through a 161-bit-wide *carry-save adder* (CSA). The carry-save output is then summed together in a carry-propagate adder to produce an unrounded result in nonredundant, two's complement form. The result is conditionally recomplemented, so as to return a result in sign-magnitude form. The complemented result is normalized, and then it is rounded to fit within the target format.

B.7 Real Stuff: The NVIDIA GeForce 8800

The NVIDIA GeForce 8800 GPU, introduced in November 2006, is a unified vertex and pixel processor design that also supports parallel computing applications written in C using the CUDA parallel programming model. It is the first implementation of the Tesla unified graphics and computing architecture described in Section B.4 and in Lindholm et al. [2008]. A family of Tesla architecture GPUs addresses the different needs of laptops, desktops, workstations, and servers.

Streaming Processor Array (SPA)

The GeForce 8800 GPU shown in Figure B.7.1 contains 128 *streaming processor* (SP) cores organized as 16 *streaming multiprocessors* (SMs). Two SMs share a texture unit in each *texture/processor cluster* (TPC). An array of eight TPCs makes up the *streaming processor array* (SPA), which executes all graphics shader programs and computing programs.

The host interface unit communicates with the host CPU via the PCI-Express bus, checks command consistency, and performs context switching. The input assembler collects geometric primitives (points, lines, triangles). The work distribution blocks dispatch vertices, pixels, and compute thread arrays to the TPCs in the SPA. The TPCs execute vertex and geometry shader programs and computing programs. Output geometric data are sent to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments that are then redistributed back into the SPA to execute pixel shader programs. Shaded pixels are sent across the interconnection network for processing by the ROP units. The network also routes texture memory read requests from the SPA to DRAM and reads data from DRAM through a level-2 cache back to the SPA.

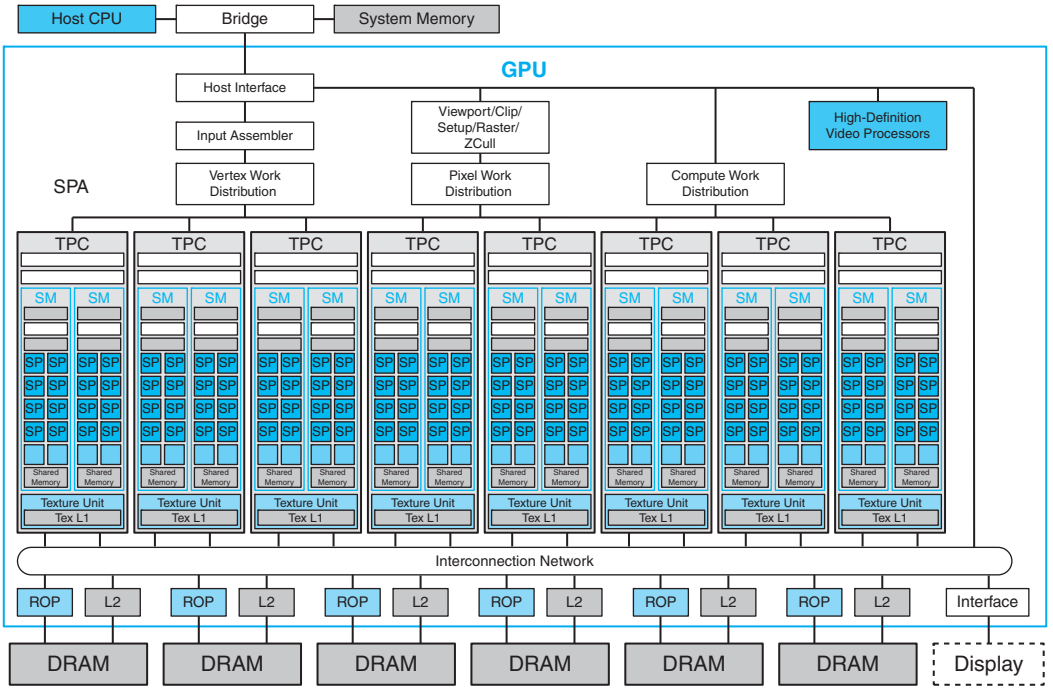


FIGURE B.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

Texture/Processor Cluster (TPC)

Each TPC contains a geometry controller, an SMC, two SMs, and a texture unit as shown in Figure B.7.2.

The geometry controller maps the logical graphics vertex pipeline into recirculation on the physical SMs by directing all primitive and vertex attribute and topology flow in the TPC.

The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simultaneously: vertex, geometry, and pixel.

The texture unit processes a texture instruction for one vertex, geometry, or pixel quad, or four compute threads per cycle. Texture instruction sources are texture coordinates, and the outputs are weighted samples, typically a four-component (RGBA) floating-point color. The texture unit is deeply pipelined. Although it contains a streaming cache to capture filtering locality, it streams hits mixed with misses without stalling.

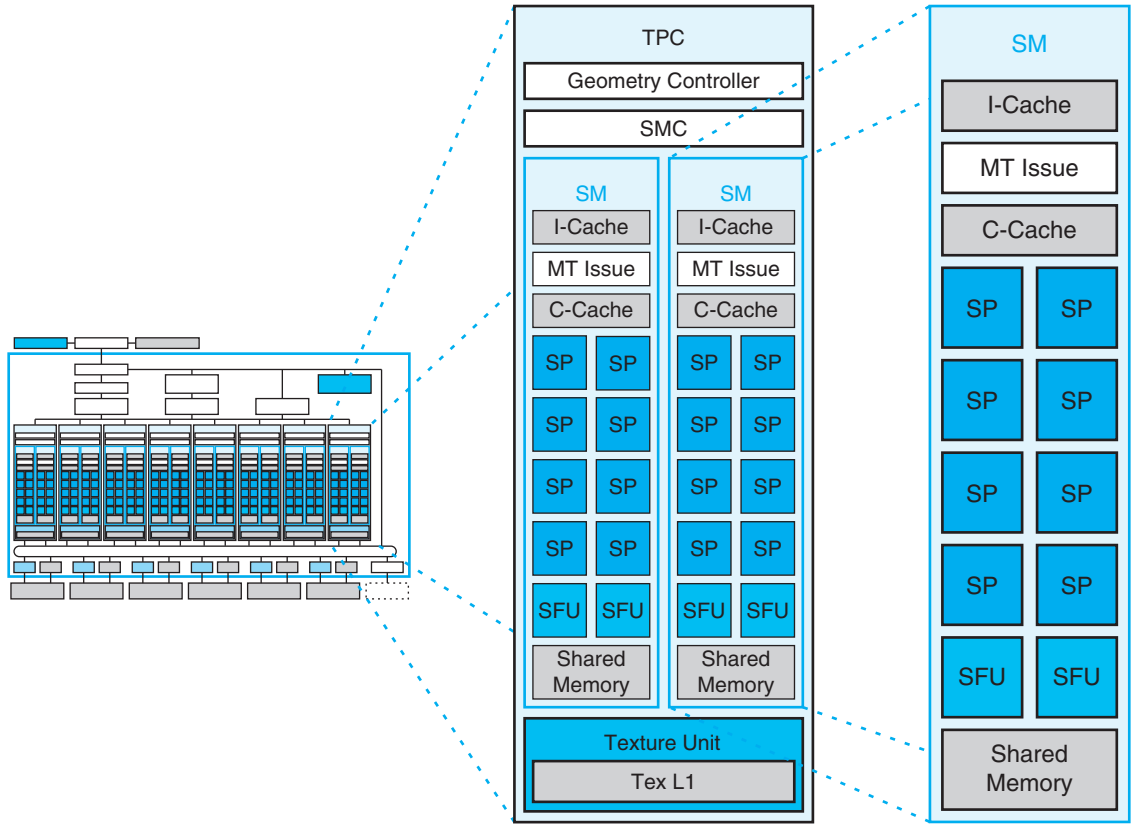


FIGURE B.7.2 Texture/processor cluster (TPC) and a streaming multiprocessor (SM). Each SM has eight *streaming processor* (SP) cores, two SFUs, and a shared memory.

Streaming Multiprocessor (SM)

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. The SM consists of eight SP thread processor cores, two SFUs, a multithreaded instruction fetch and issue unit (MT issue), an instruction cache, a read-only constant cache, and a 16 KB read/write shared memory. It executes scalar instructions for individual threads.

The GeForce 8800 Ultra clocks the SP cores and SFUs at 1.5 GHz, for a peak of 36 GFLOPS per SM. To optimize power and area efficiency, some SM nondatapath units operate at half the SP clock rate.

To efficiently execute hundreds of parallel threads while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path.

A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The SIMT design, previously described in Section B.4, shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

The SM schedules and executes multiple warp types concurrently. Each issue cycle, the scheduler selects one of the 24 warps to execute a SIMT warp instruction. An issued warp instruction executes as four sets of eight threads over four processor cycles. The SP and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and "fairness" to all warps executing in the SM.

The SM executes *cooperative thread arrays* (CTAs) as multiple concurrent warps which access a shared memory region allocated dynamically for the CTA.

Instruction Set

Threads execute scalar instructions, unlike previous GPU vector instruction architectures. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

The register-based instruction set includes all the floating-point and integer arithmetic, transcendental, logical, flow control, memory load/store, and texture instructions listed in the PTX instruction table of Figure B.4.3. Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic. For computing, the load/store instructions access three read-write memory spaces: local memory for per-thread, private, temporary data; shared memory for low-latency per-CTA data shared by the threads of the CTA; and global memory for data shared by all threads. Computing programs use the fast barrier synchronization `bar.sync` instruction to synchronize threads within a CTA that communicate with each other via shared and global memory. The latest Tesla architecture GPUs implement PTX atomic memory operations, which facilitate parallel reductions and parallel data structure management.

Streaming Processor (SP)

The multithreaded SP core is the primary thread processor, as introduced in Section B.4. Its register file provides 1024 scalar 32-bit registers for up to 96 threads (more threads than in the example SP of Section B.4). Its floating-point add and

multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including *not-a-number* (NaN) and infinity. The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions in Figure B.4.3. The processor is fully pipelined, and latency is optimized to balance delay and area.

Special Function Unit (SFU)

The SFU supports computation of both transcendental functions and planar attribute interpolation. As described in Section B.6, it uses quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x, and sin/cos functions at one result per cycle. The SFU also supports pixel attribute interpolation such as color, depth, and texture coordinates at four samples per cycle.

Rasterization

Geometry primitives from the SMs go in their original round-robin input order to the viewport/clip/setup/raster/zcull block. The viewport and clip units clip the primitives to the view frustum and to any enabled user clip planes, and then transform the vertices into screen (pixel) space.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. A coarse-rasterization stage generates all pixel tiles that are at least partially inside the primitive. The zcull unit maintains a hierarchical *z* surface, rejecting pixel tiles if they are conservatively known to be occluded by previously drawn pixels. The rejection rate is up to 256 pixels per clock. Pixels that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values.

The depth test and update can be performed ahead of the fragment shader, or after, depending on current state. The SMC assembles surviving pixels into warps to be processed by an SM running the current pixel shader. The SMC then sends surviving pixel and associated data to the ROP.

Raster Operations Processor (ROP) and Memory System

Each ROP is paired with a specific memory partition. For each pixel fragment emitted by a pixel shader program, ROPs perform depth and stencil testing and updates, and in parallel, color blending and updates. Lossless color compression (up to 8:1) and depth compression (up to 8:1) are used to reduce DRAM bandwidth. Each ROP has a peak rate of four pixels per clock and supports 16-bit floating-point and 32-bit floating-point HDR formats. ROPs support double-rate-depth processing when color writes are disabled.

Antialiasing support includes up to 16× multisampling and supersampling. The *coverage-sampling antialiasing* (CSAA) algorithm computes and stores Boolean coverage at up to 16 samples and compresses redundant color, depth, and stencil information into the memory footprint and a bandwidth of four or eight samples for improved performance.

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1.0 GHz, yielding a bandwidth of about 16 GB/s per partition, or 96 GB/s.

The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths. Texture and load/store requests can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Scalability

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right balance for different performance and cost targets in GPU market segments. *Scalable link interconnect* (SLI) connects multiple GPUs, providing further scalability.

Performance

The GeForce 8800 Ultra clocks the SP thread processor cores and SFUs at 1.5 GHz, for a theoretical operation peak of 576 GFLOPS. The GeForce 8800 GTX has a 1.35 GHz processor clock and a corresponding peak of 518 GFLOPS.

The following three sections compare the performance of a GeForce 8800 GPU with a multicore CPU on three different applications—dense linear algebra, fast Fourier transforms, and sorting. The GPU programs and libraries are compiled CUDA C code. The CPU code uses the single-precision multithreaded Intel MKL 10.0 library to leverage SSE instructions and multiple cores.

Dense Linear Algebra Performance

Dense linear algebra computations are fundamental in many applications. Volkov and Demmel [2008] present GPU and CPU performance results for single-precision dense matrix-matrix multiplication (the SGEMM routine) and LU, QR, and Cholesky matrix factorizations. Figure B.7.3 compares GFLOPS rates on SGEMM dense matrix-matrix multiplication for a GeForce 8800 GTX GPU with a quad-core CPU. Figure B.7.4 compares GFLOPS rates on matrix factorization for a GPU with a quad-core CPU.

Because SGEMM matrix-matrix multiply and similar BLAS3 routines are the bulk of the work in matrix factorization, their performance sets an upper bound on factorization rate. As the matrix order increases beyond 200 to 400, the factorization

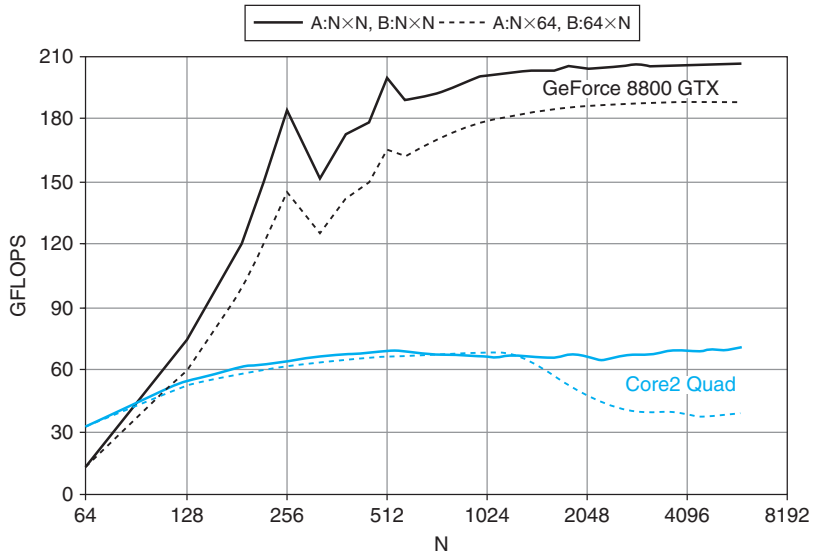


FIGURE B.7.3 SGEMM dense matrix-matrix multiplication performance rates. The graph shows single-precision GFLOPS rates achieved in multiplying square N×N matrices (solid lines) and thin N ×64 and 64×N matrices (dashed lines). Adapted from Figure 6 of Volkov and Demmel [2008]. The black lines are a 1.35 GHz GeForce 8800 GTX using Volkov's SGEMM code (now in NVIDIA CUBLAS 2.0) on matrices in GPU memory. The blue lines are a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0 on matrices in CPU memory.

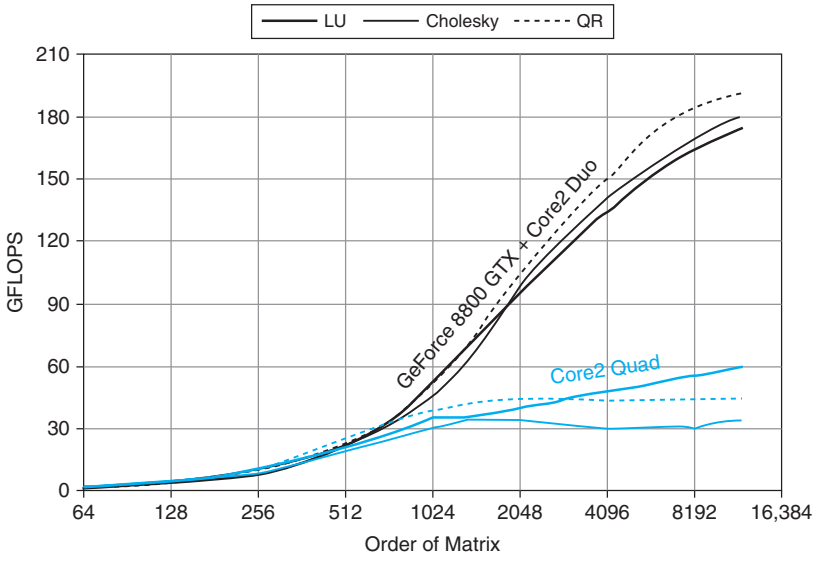


FIGURE B.7.4 Dense matrix factorization performance rates. The graph shows GFLOPS rates achieved in matrix factorizations using the GPU and using the CPU alone. Adapted from Figure 7 of Volkov and Demmel [2008]. The black lines are for a 1.35 GHz NVIDIA GeForce 8800 GTX, CUDA 1.1, Windows XP attached to a 2.67 GHz Intel Core2 Duo E6700 Windows XP, including all CPU–GPU data transfer times. The blue lines are for a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0.

problem becomes large enough that SGEMM can leverage the GPU parallelism and overcome the CPU–GPU system and copy overhead. Volkov's SGEMM matrix-matrix multiply achieves 206 GFLOPS, about 60% of the GeForce 8800 GTX peak multiply-add rate, while the QR factorization reached 192 GFLOPS, about 4.3 times the quad-core CPU.

FFT Performance

Fast Fourier Transforms (FFTs) are used in many applications. Large transforms and multidimensional transforms are partitioned into batches of smaller 1D transforms.

Figure B.7.5 compares the in-place 1D complex single-precision FFT performance of a 1.35 GHz GeForce 8800 GTX (dating from late 2006) with a 2.8 GHz quad-Core Intel Xeon E5462 series (code named "Harpertown," dating from late 2007). CPU performance was measured using the Intel *Math Kernel Library* (MKL) 10.0 FFT with four threads. GPU performance was measured using the NVIDIA CUFFT 2.1 library and batched 1D radix-16 decimation-in-frequency FFTs. Both CPU and GPU throughput performance was measured using batched FFTs; batch size was $2^{24}/n$, where n is the transform size. Thus, the workload for every transform size was 128 MB. To determine GFLOPS rate, the number of operations per transform was taken as $5n \log_2 n$.

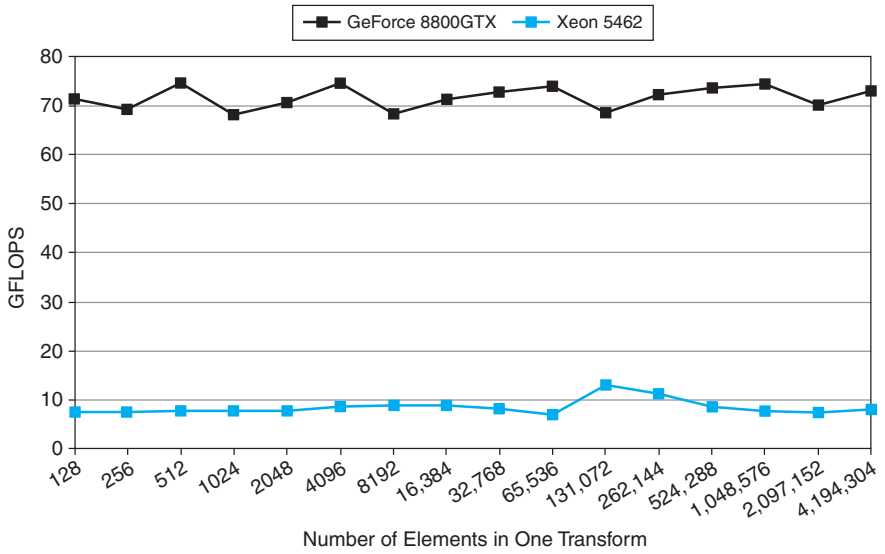


FIGURE B.7.5 Fast Fourier transform throughput performance. The graph compares the performance of batched one-dimensional in-place complex FFTs on a 1.35 GHz GeForce 8800 GTX with a quad-core 2.8 GHz Intel Xeon E5462 series (code named "Harpertown"), 6MB L2 Cache, 4GB Memory, 1600 FSB, Red Hat Linux, Intel MKL 10.0.

Sorting Performance

In contrast to the applications just discussed, sort requires far more substantial coordination among parallel threads, and parallel scaling is correspondingly harder to obtain. Nevertheless, a variety of well-known sorting algorithms can be efficiently parallelized to run well on the GPU. Satish et al. [2008] detail the design of sorting algorithms in CUDA, and the results they report for radix sort are summarized below.

Figure B.7.6 compares the parallel sorting performance of a GeForce 8800 Ultra with an 8-core Intel Clovertown system, both of which date to early 2007. The CPU cores are distributed between two physical sockets. Each socket contains a multichip module with twin Core2 chips, and each chip has a 4MB L2 cache. All sorting routines were designed to sort key-value pairs where both keys and values are 32-bit integers. The primary algorithm being studied is radix sort, although the quicksort-based `parallel_sort()` procedure provided by Intel's Threading Building Blocks is also included for comparison. Of the two CPU-based radix sort codes, one was implemented using only the scalar instruction set and the other utilizes carefully hand-tuned assembly language routines that take advantage of the SSE2 SIMD vector instructions.

The graph itself shows the achieved sorting rate—defined as the number of elements sorted divided by the time to sort—for a range of sequence sizes. It is

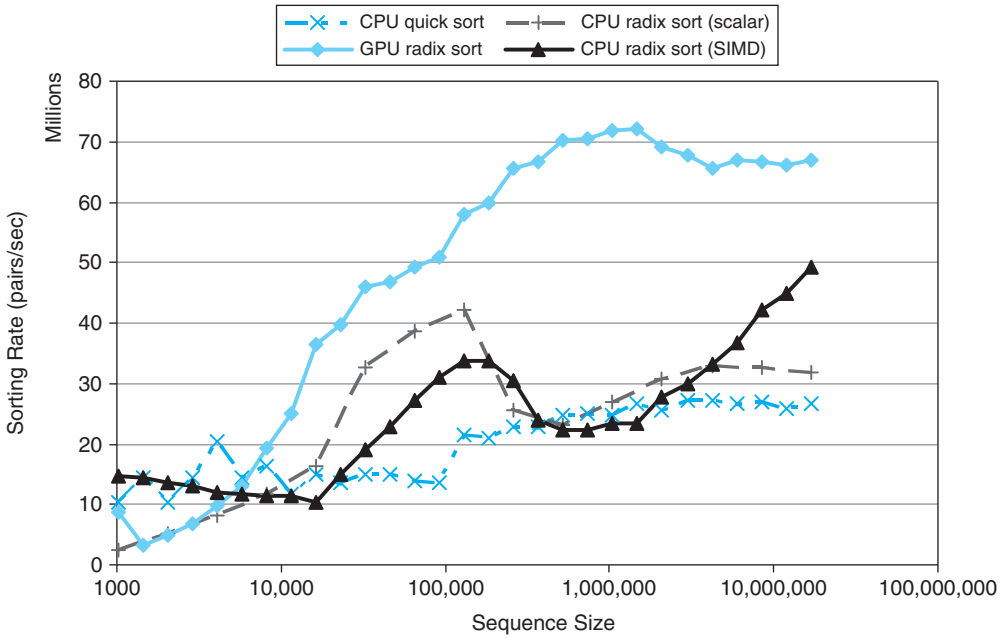


FIGURE B.7.6 Parallel sorting performance. This graph compares sorting rates for parallel radix sort implementations on a 1.5 GHz GeForce 8800 Ultra and an 8-core 2.33 GHz Intel Core2 Xeon E5345 system.

apparent from this graph that the GPU radix sort achieved the highest sorting rate for all sequences of 8K-elements and larger. In this range, it is on average 2.6 times faster than the quicksort-based routine and roughly two times faster than the radix sort routines, all of which were using the eight available CPU cores. The CPU radix sort performance varies widely, likely due to poor cache locality of its global permutations.

B.8 Real Stuff: Mapping Applications to GPUs

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop mainstream visual computing and high-performance computing applications that transparently scale their parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to GPUs with widely varying numbers of cores.

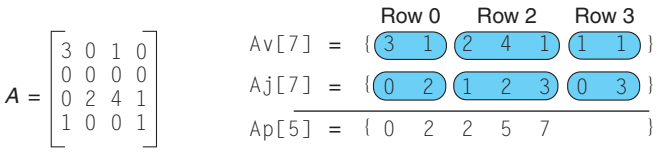
This section presents examples of mapping scalable parallel computing applications to the GPU using CUDA.

Sparse Matrices

A wide variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. *Sparse matrix-vector multiplication* (SpMV) is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss below, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient method straightforward.

A sparse $n \times n$ matrix is one in which the number of nonzero entries m is only a small fraction of the total. Sparse matrix representations seek to store only the nonzero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only $m = O(n)$ nonzero elements, this represents a substantial saving in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the *compressed sparse row* (CSR) representation. The m nonzero elements of the matrix A are stored in row-major order in an array `Av`. A second array `Aj` records the corresponding column index for each entry of `Av`. Finally, an array `Ap` of $n+1$ elements records the extent of each row in the previous arrays; the entries for row i in `Aj` and `Av` extend from index `Ap[i]` up to, but not including, index `Ap[i + 1]`. This implies that `Ap[0]` will always be 0 and `Ap[n]` will always be the number of nonzero elements in the matrix. Figure B.8.1 shows an example of the CSR representation of a simple matrix.



a. Sample matrix *A* b. CSR representation of matrix

FIGURE B.8.1 Compressed sparse row (CSR) matrix.

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // column indices for row
                   float *Av,        // nonzero entries for row
                   float *x)         // the RHS vector
{
    float sum = 0;

    for(unsigned int column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
```

FIGURE B.8.2 Serial C code for a single row of sparse matrix-vector multiply.

Given a matrix A in CSR form and a vector x, we can compute a single row of the product $y = Ax$ using the `multiply_row()` procedure shown in Figure B.8.2. Computing the full product is then simply a matter of looping over all rows and computing the result for that row using `multiply_row()`, as in the serial C code shown in Figure B.8.3.

This algorithm can be translated into a parallel CUDA kernel quite easily. We simply spread the loop in `csrmul_serial()` over many parallel threads. Each thread will compute exactly one row of the output vector y. The code for this kernel is shown in Figure B.8.4. Note that it looks extremely similar to the serial loop used in the `csrmul_serial()` procedure. There are really only two points of difference. First, the row index for each thread is computed from the block and thread indices assigned to each thread, eliminating the for-loop. Second, we have a conditional that only evaluates a row product if the row index is within the bounds of the matrix (this is necessary since the number of rows n need not be a multiple of the block size used in launching the kernel).

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    for(unsigned int row=0; row<num_rows; ++row)
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

FIGURE B.8.3 Serial code for sparse matrix-vector multiply.

```
__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

FIGURE B.8.4 CUDA version of sparse matrix-vector multiply.

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like:

```
unsigned int blocksize = 128;  // or any size up to 512
unsigned int nblocks = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

The pattern that we see here is a very common one. The original serial algorithm is a loop whose iterations are independent of each other. Such loops can be parallelized quite easily by simply assigning one or more iterations of the loop to each parallel thread. The programming model provided by CUDA makes expressing this type of parallelism particularly straightforward.

This general strategy of decomposing computations into blocks of independent work, and more specifically breaking up independent loop iterations, is not unique to CUDA. This is a common approach used in one form or another by various parallel programming systems, including OpenMP and Intel's Threading Building Blocks.

Caching in Shared Memory

The SpMV algorithms outlined above are fairly simplistic. There are a number of optimizations that can be made in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking. The parallel kernels can also be reimplemented in terms of data parallel *scan* operations presented by Sengupta et al. [2007].

One of the important architectural features exposed by CUDA is the presence of the per-block shared memory, a small on-chip memory with very low latency. Taking advantage of this memory can deliver substantial performance improvements. One common way of doing this is to use shared memory as a software-managed cache to hold frequently reused data. Modifcations using shared memory are shown in Figure B.8.5.

In the context of sparse matrix multiplication, we observe that several rows of A may use a particular array element `x[i]`. In many common cases, and particularly when the matrix has been reordered, the rows using `x[i]` will be rows near row i. We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows i through j will load `x[i]` through `x[j]` into its shared memory. We will unroll the `multiply_row()` loop and fetch elements of `x` from the cache whenever possible. The resulting code is shown in Figure B.8.5. Shared memory can also be used to make other optimizations, such as fetching `Ap[row+1]` from an adjacent thread rather than refetching it from memory.

Because the Tesla architecture provides an explicitly managed on-chip shared memory, rather than an implicitly active hardware cache, it is fairly common to add this sort of optimization. Although this can impose some additional development burden on the programmer, it is relatively minor, and the potential performance benefits can be substantial. In the example shown above, even this fairly simple use of shared memory returns a roughly 20% performance improvement on representative matrices derived from 3D surface meshes. The availability of an explicitly managed memory in lieu of an implicit cache also has the advantage that caching and prefetching policies can be specifically tailored to the application needs.

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
    // Cache the rows of x[] corresponding to this block.
    __shared__ float cache[blocksize];

    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end   = block_begin + blockDim.x;
    unsigned int row         = block_begin + threadIdx.x;

    // Fetch and cache our window of x[].
    if( row<num_rows)  cache[threadIdx.x] = x[row];
    __syncthreads();

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];
        float sum = 0, x_j;

        for(unsigned int col=row_begin; col<row_end; ++col)
        {
            unsigned int j = Aj[col];

            // Fetch x_j from our cache when possible
            if( j>=block_begin && j<block_end )
                x_j = cache[j-block_begin];
            else
                x_j = x[j];

            sum += Av[col] * x_j;
        }

        y[row] = sum;
    }
}
```

FIGURE B.8.5 Shared memory version of sparse matrix-vector multiply.

These are fairly simple kernels whose purpose is to illustrate basic techniques in writing CUDA programs, rather than how to achieve maximal performance. Numerous possible avenues for optimization are available, several of which are explored by Williams et al. [2007] on a handful of different multicore architectures. Nevertheless, it is still instructive to examine the comparative performance of even these simplistic kernels. On a 2 GHz Intel Core2 Xeon E5335 processor, the `csrmul_serial()` kernel runs at roughly 202 million nonzeros processed per second, for a collection of Laplacian matrices derived from 3D triangulated surface meshes. Parallelizing this kernel with the `parallel_for` construct provided by Intel's Threading Building Blocks produces parallel speed-ups of 2.0, 2.1, and 2.3 running on two, four, and eight cores of the machine, respectively. On a GeForce 8800 Ultra, the `csrmul_kernel()` and `csrmul_cached()` kernels achieve processing rates of roughly 772 and 920 million nonzeros per second, corresponding to parallel speed-ups of 3.8 and 4.6 times over the serial performance of a single CPU core.

Scan and Reduction

Parallel *scan*, also known as parallel *prefix sum*, is one of the most important building blocks for data-parallel algorithms [Blelloch, 1990]. Given a sequence a of n elements:

$$[a_0, a_1, \ldots, a_{n-1}]$$

and a binary associative operator $\oplus$, the `scan` function computes the sequence:

$$\text{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

As an example, if we take $\oplus$ to be the usual addition operator, then applying scan to the input array

$$a = [3\,1\,7\,0\,4\,1\,6\,3]$$

will produce the sequence of partial sums:

$$\text{scan}(a, +) = [3\,4\,11\,11\,15\,16\,22\,25]$$

This scan operator is an *inclusive* scan, in the sense that element i of the output sequence incorporates element a_i of the input. Incorporating only previous elements would yield an *exclusive* scan operator, also known as a *prefix-sum* operation.

The serial implementation of this operation is extremely simple. It is simply a loop that iterates once over the entire sequence, as shown in Figure B.8.6.

At first glance, it might appear that this operation is inherently serial. However, it can actually be implemented in parallel efficiently. The key observation is that

```
template<class T>
__host__ T plus_scan(T *x, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        x[i] = x[i-1] + x[i];
}
```

FIGURE B.8.6 Template for serial plus-scan.

```
template<class T>
__device__ T plus_scan(T *x)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset)  t = x[i-offset];
        __syncthreads();

        if(i>=offset)  x[i] = t + x[i];
        __syncthreads();
    }
    return x[i];
}
```

FIGURE B.8.7 CUDA template for parallel plus-scan.

because addition is associative, we are free to change the order in which elements are added together. For instance, we can imagine adding pairs of consecutive elements in parallel, and then adding these partial sums, and so on.

One simple scheme for doing this is from Hillis and Steele [1989]. An implementation of their algorithm in CUDA is shown in Figure B.8.7. It assumes that the input array `x[ ]` contains exactly one element per thread of the thread block. It performs $\log_2 n$ iterations of a loop collecting partial sums together.

To understand the action of this loop, consider Figure B.8.8, which illustrates the simple case for n=8 threads and elements. Each level of the diagram represents one step of the loop. The lines indicate the location from which the data are being fetched. For each element of the output (i.e., the final row of the diagram) we are building a summation tree over the input elements. The edges highlighted in blue show the form of this summation tree for the final element. The leaves of this tree are all the initial elements. Tracing back from any output element shows that it incorporates all input values up to and including itself.

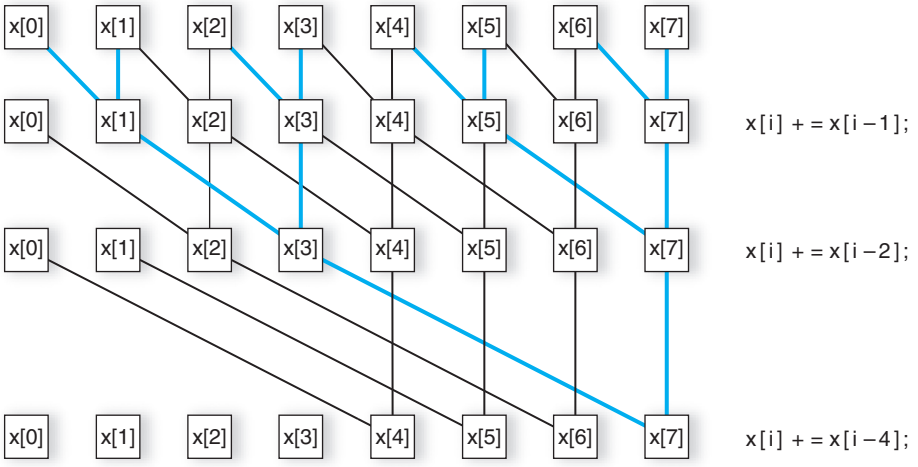


FIGURE B.8.8 Tree-based parallel scan data references.

While simple, this algorithm is not as efficient as we would like. Examining the serial implementation, we see that it performs *O(n)* additions. The parallel implementation, in contrast, performs *O(n* log *n)* additions. For this reason, it is not *work efficient*, since it does more work than the serial implementation to compute the same result. Fortunately, there are other techniques for implementing scan that are work-efficient. Details on more efficient implementation techniques and the extension of this per-block procedure to multiblock arrays are provided by Sengupta et al. [2007].

In some instances, we may only be interested in computing the sum of all elements in an array, rather than the sequence of all prefix sums returned by `scan`. This is the *parallel reduction* problem. We could simply use a scan algorithm to perform this computation, but reduction can generally be implemented more efficiently than scan.

Figure B.8.9 shows the code for computing a reduction using addition. In this example, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length 1). At the end of the reduction, we want thread 0 to hold the sum of all elements initially loaded by the threads of its block. The loop in this kernel implicitly builds a summation tree over the input elements, much like the scan algorithm above.

At the end of this loop, thread 0 holds the sum of all the values loaded by this block. If we want the final value of the location pointed to by `total` to contain the total of all elements in the array, we must combine the partial sums of all the blocks in the grid. One strategy to do this would be to have each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until we had reduced the sequence to a single value. A more attractive alternative supported by the Tesla GPU architecture is to use the `atomicAdd()` primitive, an efficient atomic

```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory, padding
    // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s)  x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 now holds the sum of all input values
    // to this block. Have it add that sum to the running total
    if( tid == 0 )  atomicAdd(total, x[tid]);
}
```

FIGURE B.8.9 CUDA implementation of plus-reduction.

read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

Parallel reduction is an essential primitive for parallel programming and highlights the importance of per-block shared memory and low-cost barriers in making cooperation among threads efficient. This degree of data shuffling among threads would be prohibitively expensive if done in off-chip global memory.

Radix Sort

One important application of scan primitives is in the implementation of sorting routines. The code in Figure B.8.10 implements a radix sort of integers across a single thread block. It accepts as input an array `values` containing one 32-bit integer for each thread of the block. For efficiency, this array should be stored in per-block shared memory, but this is not required for the sort to behave correctly.

This is a fairly simple implementation of radix sort. It assumes the availability of a procedure `partition_by_bit()` that will partition the given array such that

```
__device__ void radix_sort(unsigned int *values)
{
    for(int bit=0; bit<32; ++bit)
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}
```

FIGURE B.8.10 CUDA code for radix sort.

```
__device__ void partition_by_bit(unsigned int *values,
                                 unsigned int bit)
{
    unsigned int i    = threadIdx.x;
    unsigned int size = blockDim.x;
    unsigned int x_i  = values[i];
    unsigned int p_i  = (x_i >> bit) & 1;

    values[i] = p_i;
    __syncthreads();

    // Compute number of T bits up to and including p_i.
    // Record the total number of F bits as well.
    unsigned int T_before = plus_scan(values);
    unsigned int T_total  = values[size-1];
    unsigned int F_total  = size - T_total;
    __syncthreads();

    // Write every x_i to its proper place
    if( p_i )
        values[T_before-1 + F_total] = x_i;
    else
        values[i - T_before] = x_i;
}
```

FIGURE B.8.11 CUDA code to partition data on a bit-by-bit basis, as part of radix sort.

all values with a 0 in the designated bit will come before all values with a 1 in that bit. To produce the correct output, this partitioning must be stable.

Implementing the partitioning procedure is a simple application of scan. Thread i holds the value x_i and must calculate the correct output index at which to write this value. To do so, it needs to calculate (1) the number of threads $j < i$ for which the designated bit is 1 and (2) the total number of bits for which the designated bit is 0. The CUDA code for `partition_by_bit()` is shown in Figure B.8.11.

A similar strategy can be applied for implementing a radix sort kernel that sorts an array of large length, rather than just a one-block array. The fundamental step remains the scan procedure, although when the computation is partitioned across multiple kernels, we must double-buffer the array of values rather than doing the partitioning in place. Details on performing radix sorts on large arrays efficiently are provided by Satish et al. [2008].

N-Body Applications on a GPU[1]

Nyland et al. [2007] describe a simple yet useful computational kernel with excellent GPU performance—the *all-pairs N-body* algorithm. It is a time-consuming component of many scientific applications. N-body simulations calculate the evolution of a system of bodies in which each body continuously interacts with every other body. One example is an astrophysical simulation in which each body represents an individual star, and the bodies gravitationally attract each other. Other examples are protein folding, where N-body simulation is used to calculate electrostatic and van der Waals forces; turbulent fluid flow simulation; and global illumination in computer graphics.

The all-pairs N-body algorithm calculates the total force on each body in the system by computing each pair-wise force in the system, summing for each body. Many scientists consider this method to be the most accurate, with the only loss of precision coming from the floating-point hardware operations. The drawback is its $O(n^2)$ computational complexity, which is far too large for systems with more than 10 bodies. To overcome this high cost, several simplifications have been proposed to yield $O(n \log n)$ and $O(n)$ algorithms; examples are the Barnes-Hut algorithm, the Fast Multipole Method and Particle-Mesh-Ewald summation. All of the *fast* methods still rely on the all-pairs method as a kernel for accurate computation of short-range forces; thus it continues to be important.

N-Body Mathematics

For gravitational simulation, calculate the body-body force using elementary physics. Between two bodies indexed by i and j, the 3D force vector is:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\| \mathbf{r}_{ij} \|^2} \times \frac{\mathbf{r}_{ij}}{\| \mathbf{r}_{ij} \|}$$

The force magnitude is calculated in the left term, while the direction is computed in the right (unit vector pointing from one body to the other).

Given a list of interacting bodies (an entire system or a subset), the calculation is simple: for all pairs of interactions, compute the force and sum for each body. Once the total forces are calculated, they are used to update each body's position and velocity, based on the previous position and velocity. The calculation of the forces has complexity $O(n^2)$, while the update is $O(n)$.

[1] Adapted from Nyland et al. [2007], "Fast N-Body Simulation with CUDA," Chapter 31 of *GPU Gems 3*.

The serial force-calculation code uses two nested for-loops iterating over pairs of bodies. The outer loop selects the body for which the total force is being calculated, and the inner loop iterates over all the bodies. The inner loop calls a function that computes the pair-wise force, then adds the force into a running sum.

To compute the forces in parallel, we assign one thread to each body, since the calculation of force on each body is independent of the calculation on all other bodies. Once all of the forces are computed, the positions and velocities of the bodies can be updated.

The code for the serial and parallel versions is shown in Figure B.8.12 and Figure B.8.13. The serial version has two nested for-loops. The conversion to CUDA, like many other examples, converts the serial outer loop to a per-thread kernel where each thread computes the total force on a single body. The CUDA kernel computes a global thread ID for each thread, replacing the iterator variable of the serial outer loop. Both kernels finish by storing the total acceleration in a global array used to compute the new position and velocity values in a subsequent step. The outer loop is replaced by a CUDA kernel grid that launches *N* threads, one for each body.

```
void accel_on_all_bodies()
{
    int i, j;
    float3 acc(0.0f, 0.0f, 0.0f);

    for (i = 0; i < N; i++) {
       for (j = 0; j < N; j++) {
          acc = body_body_interaction(acc, body[i], body[j]);
       }
       accel[i] = acc;
    }
}
```

FIGURE B.8.12 Serial code to compute all pair-wise forces on N bodies.

```
__global__ void accel_on_one_body()
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
      int j;
      float3 acc(0.0f, 0.0f, 0.0f);

      for (j = 0; j < N; j++) {
          acc = body_body_interaction(acc, body[i], body[j]);
      }
      accel[i] = acc;
}
```

FIGURE B.8.13 CUDA thread code to compute the total force on a single body.

Optimization for GPU Execution

The CUDA code shown is functionally correct, but is not efficient, as it ignores key architectural features. Better performance can be achieved with three main optimizations. First, shared memory can be used to avoid identical memory reads between threads. Second, using multiple threads per body improves performance for small values of *N*. Third, loop unrolling reduces loop overhead.

Using Shared Memory

Shared memory can hold a subset of body positions, much like a cache, eliminating redundant global memory requests between threads. We optimize the code shown above to have each of *p* threads in a thread-block load *one* position into shared memory (for a total of *p* positions). Once all the threads have loaded a value into shared memory, ensured by `__syncthreads()`, each thread can then perform *p* interactions (using the data in shared memory). This is repeated *N*/*p* times to complete the force calculation for each body, which reduces the number of requests to memory by a factor of *p* (typically in the range 32–128).

The function called `accel_on_one_body()` requires a few changes to support this optimization. The modified code is shown in Figure B.8.14.

```
__shared__ float4 shPosition[256];
…
__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j, k;
    int p = blockDim.x;
    float3 acc(0.0f, 0.0f, 0.0f);
    float4 myBody = body[i];

    for (j = 0; j < N; j += p) {  // Outer loops jumps by p each time
        shPosition[threadIdx.x] = body[j+threadIdx.x];
        __syncthreads();
        for (k = 0; k < p; k++) { // Inner loop accesses p positions
            acc = body_body_interaction(acc, myBody, shPosition[k]);
        }
        __syncthreads();
    }
    accel[i] = acc;
}
```

FIGURE B.8.14 CUDA code to compute the total force on each body, using shared memory to improve performance.

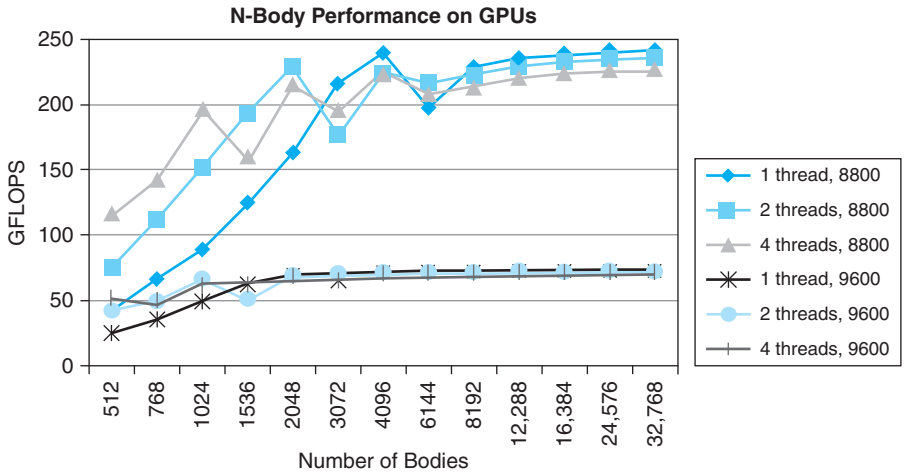


FIGURE B.8.15 Performance measurements of the N-body application on a GeForce 8800 GTX and a GeForce 9600. The 8800 has 128 stream processors at 1.35 GHz, while the 9600 has 64 at 0.80 GHz (about 30% of the 8800). The peak performance is 242 GFLOPS. For a GPU with more processors, the problem needs to be bigger to achieve full performance (the 9600 peak is around 2048 bodies, while the 8800 doesn't reach its peak until 16,384 bodies). For small N, more than one thread per body can significantly improve performance, but eventually incurs a performance penalty as N grows.

The loop that formerly iterated over all bodies now jumps by the block dimension p. Each iteration of the outer loop loads p successive positions into shared memory (one position per thread). The threads synchronize, and then p force calculations are computed by each thread. A second synchronization is required to ensure that new values are not loaded into shared memory prior to all threads completing the force calculations with the current data.

Using shared memory reduces the memory bandwidth required to less than 10% of the total bandwidth that the GPU can sustain (using less than 5 GB/s). This optimization keeps the application busy performing computation rather than waiting on memory accesses, as it would have done without the use of shared memory. The performance for varying values of N is shown in Figure B.8.15.

Using Multiple Threads per Body

Figure B.8.15 shows performance degradation for problems with small values of N (N< 4096) on the GeForce 8800 GTX. Many research efforts that rely on N-body calculations focus on small N (for long simulation times), making it a target of our optimization efforts. Our presumption to explain the lower performance was that there was simply not enough work to keep the GPU busy when N is small. The solution is to allocate more threads per body. We change the thread-block dimensions from $(p, 1, 1)$ to $(p, q, 1)$, where q threads divide the work of a single body into equal parts. By allocating the additional threads within the same thread block, partial results can be stored in shared memory. When all the force calculations are

done, the q partial results can be collected and summed to compute the final result. Using two or four threads per body leads to large improvements for small N.

As an example, the performance on the 8800 GTX jumps by 110% when $N = 1024$ (one thread achieves 90 GFLOPS, where four achieve 190 GFLOPS). Performance degrades slightly on large N, so we only use this optimization for N smaller than 4096. The performance increases are shown in Figure B.8.15 for a GPU with 128 processors and a smaller GPU with 64 processors clocked at two-thirds the speed.

Performance Comparison

The performance of the N-body code is shown in Figure B.8.15 and Figure B.8.16. In Figure B.8.15, performance of high- and medium-performance GPUs is shown, along with the performance improvements achieved by using multiple threads per body. The performance on the faster GPU ranges from 90 to just under 250 GFLOPS.

Figure B.8.16 shows nearly identical code (C++ versus CUDA) running on Intel Core2 CPUs. The CPU performance is about 1% of the GPU, in the range of 0.2 to 2 GFLOPS, remaining nearly constant over the wide range of problem sizes.

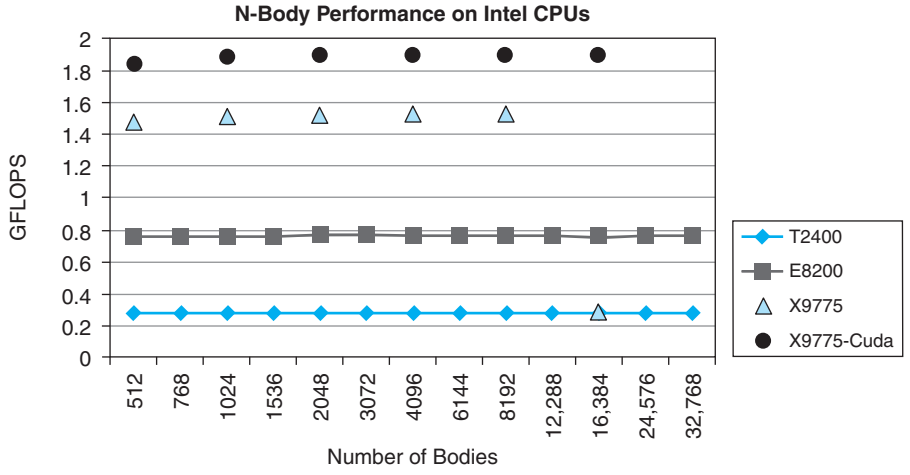


FIGURE B.8.16 Performance measurements on the N-body code on a CPU. The graph shows single precision N-body performance using Intel Core2 CPUs, denoted by their CPU model number. Note the dramatic reduction in GFLOPS performance (shown in GFLOPS on the y-axis), demonstrating how much faster the GPU is compared to the CPU. The performance on the CPU is generally independent of problem size, except for an anomalously low performance when N = 16,384 on the X9775 CPU. The graph also shows the results of running the CUDA version of the code (using the CUDA-for-CPU compiler) on a single CPU core, where it outperforms the C++ code by 24%. As a programming language, CUDA exposes parallelism and locality that a compiler can exploit. The Intel CPUs are a 3.2 GHz Extreme X9775 (code named "Penryn"), a 2.66 GHz E8200 (code named "Wolfdale"), a desktop, pre-Penryn CPU, and a 1.83 GHz T2400 (code named "Yonah"), a 2007 laptop CPU. The Penryn version of the Core 2 architecture is particularly interesting for N-body calculations with its 4-bit divider, allowing division and square root operations to execute four times faster than previous Intel CPUs.

The graph also shows the results of compiling the CUDA version of the code for a CPU, where the performance improves by 24%. CUDA, as a programming language, exposes parallelism, allowing the compiler to make better use of the SSE vector unit on a single core. The CUDA version of the N-body code naturally maps to multicore CPUs as well (with grids of blocks), where it achieves nearly perfect scaling on an eight-core system with N = 4096 (ratios of 2.0, 3.97, and 7.94 on two, four, and eight cores, respectively).

Results

With a modest effort, we developed a computational kernel that improves GPU performance over multicore CPUs by a factor of up to 157. Execution time for the N-body code running on a recent CPU from Intel (Penryn X9775 at 3.2 GHz, single core) took more than 3 seconds per frame to run the same code that runs at a 44 Hz frame rate on a GeForce 8800 GPU. On pre-Penryn CPUs, the code requires 6–16 seconds, and on older Core2 processors and Pentium IV processor, the time is about 25 seconds. We must divide the apparent increase in performance in half, as the CPU requires only half as many calculations to compute the same result (using the optimization that the forces on a pair of bodies are equal in strength and opposite in direction).

How can the GPU speed up the code by such a large amount? The answer requires inspecting architectural details. The pair-wise force calculation requires 20 floating-point operations, comprised mostly of addition and multiplication instructions (some of which can be combined using a multiply-add instruction), but there are also division and square root instructions for vector normalization. Intel CPUs take many cycles for single-precision division and square root instructions,[2] although this has improved in the latest Penryn CPU family with its faster 4-bit divider.[3] Additionally, the limitations in register capacity lead to many MOV instructions in the x86 code (presumably to/from L1 cache). In contrast, the GeForce 8800 executes a reciprocal square-root thread instruction in four clocks; see Section B.6 for special function accuracy. It has a larger register file (per thread) and shared memory that can be accessed as an instruction operand. Finally, the CUDA compiler emits 15 instructions for one iteration of the loop, compared with more than 40 instructions from a variety of x86 CPU compilers. Greater parallelism, faster execution of complex instructions, more register space, and an efficient compiler all combine to explain the dramatic performance improvement of the N-body code between the CPU and the GPU.

[2] The x86 SSE instructions reciprocal-square-root (RSQRT*) and reciprocal (RCP*) were not considered, as their accuracy is too low to be comparable.

[3] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November 2007. Order Number: 248966-016. Also available at www.intel.com/design/processor/manuals/248966.pdf.

On a GeForce 8800, the all-pairs N-body algorithm delivers more than 240 GFLOPS of performance, compared to less than 2 GFLOPS on recent sequential processors. Compiling and executing the CUDA version of the code on a CPU demonstrates that the problem scales well to multicore CPUs, but is still significantly slower than a single GPU.

We coupled the GPU N-body simulation with a graphical display of the motion, and can interactively display 16K bodies interacting at 44 frames per second. This allows astrophysical and biophysical events to be displayed and navigated at interactive rates. Additionally, we can parameterize many settings, such as noise reduction, damping, and integration techniques, immediately displaying their effects on the dynamics of the system. This provides scientists with stunning visual imagery, boosting their insights on otherwise invisible systems (too large or small, too fast or too slow), allowing them to create better models of physical phenomena.

Figure B.8.17 shows a time-series display of an astrophysical simulation of 16K bodies, with each body acting as a galaxy. The initial configuration is a spherical shell

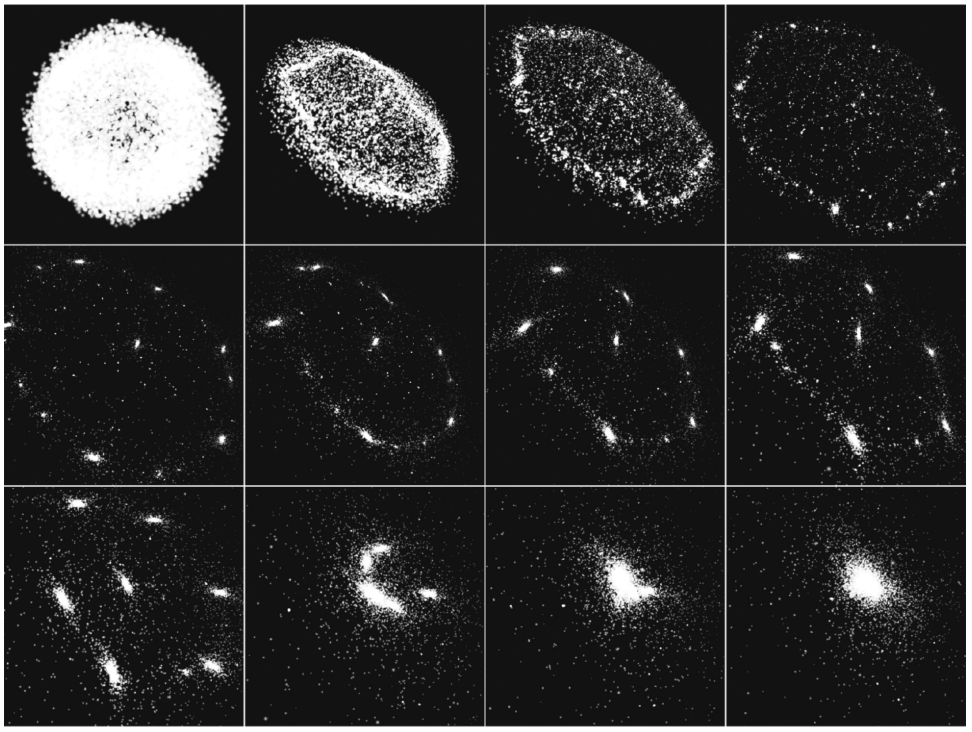


FIGURE B.8.17 Twelve images captured during the evolution of an N-body system with 16,384 bodies.

of bodies rotating about the *z*-axis. One phenomenon of interest to astrophysicists is the clustering that occurs, along with the merging of galaxies over time. For the interested reader, the CUDA code for this application is available in the CUDA SDK from www.nvidia.com/CUDA.

B.9 Fallacies and Pitfalls

GPUs have evolved and changed so rapidly that many fallacies and pitfalls have arisen. We cover a few here.

Fallacy GPUs are just SIMD vector multiprocessors.

It is easy to draw the false conclusion that GPUs are simply SIMD vector multiprocessors. GPUs do have a SPMD-style programming model, in that a programmer can write a single program that is executed in multiple thread instances with multiple data. The execution of these threads is not purely SIMD or vector, however; it is *single-instruction multiple-thread* (SIMT), described in Section B.4. Each GPU thread has its own scalar registers, thread private memory, thread execution state, thread ID, independent execution and branch path, and effective program counter, and can address memory independently. Although a group of threads (e.g., a warp of 32 threads) executes more efficiently when the PCs for the threads are the same, this is not necessary. So, the multiprocessors are not purely SIMD. The thread execution model is MIMD with barrier synchronization and SIMT optimizations. Execution is more efficient if individual thread load/store memory accesses can be coalesced into block accesses, as well. However, this is not strictly necessary. In a purely SIMD vector architecture, memory/register accesses for different threads must be aligned in a regular vector pattern. A GPU has no such restriction for register or memory accesses; however, execution is more efficient if warps of threads access local blocks of data.

In a further departure from a pure SIMD model, an SIMT GPU can execute more than one warp of threads concurrently. In graphics applications, there may be multiple groups of vertex programs, pixel programs, and geometry programs running in the multiprocessor array concurrently. Computing programs may also execute different programs concurrently in different warps.

Fallacy GPU performance cannot grow faster than Moore's law.

Moore's law is simply a rate. It is not a "speed of light" limit for any other rate. Moore's law describes an expectation that, over time, as semiconductor technology advances and transistors become smaller, the manufacturing cost per transistor will decline exponentially. Put another way, given a constant manufacturing cost, the

number of transistors will increase exponentially. Gordon Moore [1965] predicted that this progression would provide roughly two times the number of transistors for the same manufacturing cost every year, and later revised it to doubling every 2 years. Although Moore made the initial prediction in 1965 when there were just 50 components per integrated circuit, it has proved remarkably consistent. The reduction of transistor size has historically had other benefits, such as lower power per transistor and faster clock speeds at constant power.

This increasing bounty of transistors is used by chip architects to build processors, memory, and other components. For some time, CPU designers have used the extra transistors to increase processor performance at a rate similar to Moore's law, so much so that many people think that processor performance growth of two times every 18–24 months is Moore's law. In fact, it is not.

Microprocessor designers spend some of the new transistors on processor cores, improving the architecture and design, and pipelining for more clock speed. The rest of the new transistors are used for providing more cache, to make memory access faster. In contrast, GPU designers use almost none of the new transistors to provide more cache; most of the transistors are used for improving the processor cores and adding more processor cores.

GPUs get faster by four mechanisms. First, GPU designers reap the Moore's law bounty directly by applying exponentially more transistors to building more parallel, and thus faster, processors. Second, GPU designers can improve on the architecture over time, increasing the efficiency of the processing. Third, Moore's law assumes constant cost, so the Moore's law rate can clearly be exceeded by spending more for larger chips with more transistors. Fourth, GPU memory systems have increased their effective bandwidth at a pace nearly comparable to the processing rate, by using faster memories, wider memories, data compression, and better caches. The combination of these four approaches has historically allowed GPU performance to double regularly, roughly every 12 to 18 months. This rate, exceeding the rate of Moore's law, has been demonstrated on graphics applications for approximately 10 years and shows no sign of significant slowdown. The most challenging rate limiter appears to be the memory system, but competitive innovation is advancing that rapidly too.

Fallacy GPUs only render 3D graphics; they can't do general computation.

GPUs are built to render 3D graphics as well as 2D graphics and video. To meet the demands of graphics software developers as expressed in the interfaces and performance/feature requirements of the graphics APIs, GPUs have become massively parallel programmable floating-point processors. In the graphics domain, these processors are programmed through the graphics APIs and with arcane graphics programming languages (GLSL, Cg, and HLSL, in OpenGL and Direct3D). However, there is nothing preventing GPU architects from exposing

the parallel processor cores to programmers without the graphics API or the arcane graphics languages.

In fact, the Tesla architecture family of GPUs exposes the processors through a software environment known as CUDA, which allows programmers to develop general application programs using the C language and soon C++. GPUs are Turing-complete processors, so they can run any program that a CPU can run, although perhaps less well. And perhaps faster.

Fallacy GPUs cannot run double-precision floating-point programs fast.

In the past, GPUs could not run double-precision floating-point programs at all, except through software emulation. And that's not very fast at all. GPUs have made the progression from indexed arithmetic representation (lookup tables for colors) to 8-bit integers per color component, to fixed-point arithmetic, to single-precision floating-point, and recently added double precision. Modern GPUs perform virtually all calculations in single-precision IEEE floating-point arithmetic, and are beginning to use double precision in addition.

For a small additional cost, a GPU can support double-precision floating-point as well as single-precision floating-point. Today, double-precision runs more slowly than the single-precision speed, about five to ten times slower. For incremental additional cost, double-precision performance can be increased relative to single precision in stages, as more applications demand it.

Fallacy GPUs don't do floating-point correctly.

GPUs, at least in the Tesla architecture family of processors, perform single-precision floating-point processing at a level prescribed by the IEEE 754 floating-point standard. So, in terms of accuracy, GPUs are the equal of any other IEEE 754-compliant processors.

Today, GPUs do not implement some of the specific features described in the standard, such as handling denormalized numbers and providing precise floating-point exceptions. However, the recently introduced Tesla T10P GPU provides full IEEE rounding, fused-multiply-add, and denormalized number support for double precision.

Pitfall Just use more threads to cover longer memory latencies.

CPU cores are typically designed to run a single thread at full speed. To run at full speed, every instruction and its data need to be available when it is time for that instruction to run. If the next instruction is not ready or the data required for that instruction is not available, the instruction cannot run and the processor stalls. External memory is distant from the processor, so it takes many cycles of wasted execution to fetch data from memory. Consequently, CPUs require large local

caches to keep running without stalling. Memory latency is long, so it is avoided by striving to run in the cache. At some point, program working set demands may be larger than any cache. Some CPUs have used multithreading to tolerate latency, but the number of threads per core has generally been limited to a small number.

The GPU strategy is different. GPU cores are designed to run many threads concurrently, but only one instruction from any thread at a time. Another way to say this is that a GPU runs each thread slowly, but in aggregate runs the threads efficiently. Each thread can tolerate some amount of memory latency, because other threads can run.

The downside of this is that multiple—many multiple threads—are required to cover the memory latency. In addition, if memory accesses are scattered or not correlated among threads, the memory system will get progressively slower in responding to each individual request. Eventually, even the multiple threads will not be able to cover the latency. So, the pitfall is that for the "just use more threads" strategy to work for covering latency, you have to have enough threads, and the threads have to be well-behaved in terms of locality of memory access.

Fallacy O(n) algorithms are difficult to speed up.

No matter how fast the GPU is at processing data, the steps of transferring data to and from the device may limit the performance of algorithms with O(n) complexity (with a small amount of work per datum). The highest transfer rate over the PCIe bus is approximately 48 GB/second when DMA transfers are used, and slightly less for nonDMA transfers. The CPU, in contrast, has typical access speeds of 8–12 GB/second to system memory. Example problems, such as vector addition, will be limited by the transfer of the inputs to the GPU and the returning output from the computation.

There are three ways to overcome the cost of transferring data. First, try to leave the data on the GPU for as long as possible, instead of moving the data back and forth for different steps of a complicated algorithm. CUDA deliberately leaves data alone in the GPU between launches to support this.

Second, the GPU supports the concurrent operations of copy-in, copy-out and computation, so data can be streamed in and out of the device while it is computing. This model is useful for any data stream that can be processed as it arrives. Examples are video processing, network routing, data compression/decompression, and even simpler computations such as large vector mathematics.

The third suggestion is to use the CPU and GPU together, improving performance by assigning a subset of the work to each, treating the system as a heterogeneous computing platform. The CUDA programming model supports allocation of work to one or more GPUs along with continued use of the CPU without the use of threads (via asynchronous GPU functions), so it is relatively simple to keep all GPUs and a CPU working concurrently to solve problems even faster.

B.10 Concluding Remarks

GPUs are massively parallel processors and have become widely used, not only for 3D graphics, but also for many other applications. This wide application was made possible by the evolution of graphics devices into programmable processors. The graphics application programming model for GPUs is usually an API such as DirectX™ or OpenGL™. For more general-purpose computing, the CUDA programming model uses an SPMD (*single-program multiple data*) style, executing a program with many parallel threads.

GPU parallelism will continue to scale with Moore's law, mainly by increasing the number of processors. Only the parallel programming models that can readily scale to hundreds of processor cores and thousands of threads will be successful in supporting manycore GPUs and CPUs. Also, only those applications that have many largely independent parallel tasks will be accelerated by massively parallel manycore architectures.

Parallel programming models for GPUs are becoming more flexible, for both graphics and parallel computing. For example, CUDA is evolving rapidly in the direction of full C/C++ functionality. Graphics APIs and programming models will likely adapt parallel computing capabilities and models from CUDA. Its SPMD-style threading model is scalable, and is a convenient, succinct, and easily learned model for expressing large amounts of parallelism.

Driven by these changes in the programming models, GPU architecture is in turn becoming more flexible and more programmable. GPU fixed-function units are becoming accessible from general programs, along the lines of how CUDA programs already use texture intrinsic functions to perform texture lookups using the GPU texture instruction and texture unit.

GPU architecture will continue to adapt to the usage patterns of both graphics and other application programmers. GPUs will continue to expand to include more processing power through additional processor cores, as well as increasing the thread and memory bandwidth available for programs. In addition, the programming models must evolve to include programming heterogeneous manycore systems including both GPUs and CPUs.

Acknowledgments

This appendix is the work of several authors at NVIDIA. We gratefully acknowledge the significant contributions of Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman, and Vasily Volkov.

B.11 Historical Perspective and Further Reading

Graphics Pipeline Evolution

3D graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then to PC accelerators in the mid- to late-1990s. During this period, three major transitions occurred:

- Performance-leading graphics subsystems declined in price from $50,000 to $200.
- Performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second.
- Native hardware capabilities evolved from wireframe (polygon outlines) to flat shaded (constant color) filled polygons, to smooth shaded (interpolated color) filled polygons, to full-scene anti-aliasing with texture mapping and rudimentary multitexturing.

Fixed-Function Graphics Pipelines

Throughout this period, graphics hardware was configurable, but not programmable by the application developer. With each generation, incremental improvements were offered. But developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The NVIDIA GeForce 3, described by Lindholm et al. [2001], took the first step toward true general shader programmability. It exposed to the application developer what had been the private internal instruction set of the floating-point vertex engine. This coincided with the release of Microsoft's DirectX 8 and OpenGL's vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating point capability to the pixel fragment stage, and made texture available at the vertex stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel fragment processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. This was part of a general trend toward unifying the functionality of the different stages, at least as far as the application programmer was concerned. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs and separate hardware dedicated to the vertex and to the fragment processing. The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

Evolution of Programmable Real-Time Graphics

During the last 30 years, graphics architecture has evolved from a simple pipeline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery that appears three-dimensional. Concurrently, many of the calculations involved became far more sophisticated and user-programmable.

In these graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the position of triangle vertexes or generating pixel colors. This data independence is a key difference between GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have 1 million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms and have evolved to be programmable. Vertex programs map the position of triangle vertices on to the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point *red, green, blue, alpha* (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. For all three types of graphics shaders, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

Between these programmable graphics pipeline stages are dozens of fixed-function stages which perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. Together, the mix of programmable and fixed-function stages is engineered to balance extreme performance with user control over the rendering algorithms.

Common rendering algorithms perform a single pass over input primitives and access other memory resources in a highly coherent manner; these algorithms provide excellent bandwidth utilization and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute-limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. Whereas CPU die area is dominated by cache memory, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (since latency can be readily hidden by a high thread count); indeed, bandwidth is typically many times higher than a CPU, exceeding 100 GB/second in some cases. The far-higher number of fine-grained lightweight threads effectively exploits the rich parallelism available.

Beginning with NVIDIA's GeForce 8800 GPU in 2006, the three programmable graphics stages are mapped to an array of unified processors; the logical graphics pipeline is physically a recirculating path that visits these processors three times, with much fixed-function graphics logic between visits. Since different rendering algorithms present wildly different loads among the three programmable stages, this unification provides processor load balancing.

Unified Graphics and Computing Processors

By the DirectX 10 generation, the functionality of vertex and pixel fragment shaders was to be made identical to the programmer, and in fact a new logical stage was introduced, the geometry shader, to process all the vertices of a primitive rather than vertices in isolation. The GeForce 8800 was designed with DirectX 10 in mind. Developers were coming up with more sophisticated shading algorithms, and this motivated a sharp increase in the available shader operation rate, particularly floating-point operations. NVIDIA chose to pursue a processor design with higher operating frequency than standard-cell methodologies had allowed, to deliver the desired operation throughput as area-efficiently as possible. High-clock-speed design requires substantially more engineering effort, and this favored designing one processor, rather than two (or three, given the new geometry stage). It became worthwhile to take on the engineering challenges of a unified processor (load balancing and recirculation of a logical pipeline onto threads of the processor array) to get the benefits of one processor design.

GPGPU: an Intermediate Step

As DirectX 9-capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and began to explore the use of GPUs to solve complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations. For example, to run many simultaneous instances of a pixel shader, a triangle had to be issued to the GPU (with clipping to a rectangle shape if that's what was desired). Shaders did not have the means to perform arbitrary scatter operations to memory. The only way to write a result to memory was to emit it as a pixel color value, and configure the framebuffer operation stage to write (or blend, if desired) the result to a two-dimensional framebuffer. Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel framebuffer, then use that framebuffer as a texture map as input to the pixel fragment shader of the next stage of the computation. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU" for general purpose computing on GPUs.

GPU Computing

While developing the Tesla architecture for the GeForce 8800, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU as a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX 10 generation, NVIDIA had already begun work on a high-efficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. This processor was designed to take advantage of the common case of groups of threads executing the same code path. NVIDIA added memory load and store instructions with integer byte addressing to support the requirements of compiled C programs. It introduced the thread block (cooperative thread array), grid of thread blocks, and barrier synchronization to dispatch and manage highly parallel computing work. Atomic memory operations were added. NVIDIA developed the CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. Workstation graphics systems gave customers a choice in pixel horsepower by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s PC graphics scaling was almost nonexistent. There was one option—the VGA controller. As 3D-capable accelerators appeared, the market had room for a range of offerings. 3dfx introduced multiboard scaling with the original SLI (*Scan Line Interleave*) on their Voodoo2, which held the performance crown for its time (1998). Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high-performance) and Vanta (low-cost), first by speed binning and packaging, then with separate chip designs (GeForce 2 GTS & GeForce 2 MX). At present, for a given architecture generation, four or five separate GPU chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx, NVIDIA continued the multi-GPU SLI concept in 2004, starting with GeForce 6800—providing multi-GPU scalability transparently to the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

CPUs are scaling to higher transistor counts by increasing the number of constant-performance cores on a die, rather than increasing the performance of a single core. At this writing the industry is transitioning from dual-core to quad-core, with eight-core not far behind. Programmers are forced to find fourfold to eightfold task parallelism to fully utilize these processors, and applications using task parallelism must be rewritten frequently to target each successive doubling of

core count. In contrast, the highly multithreaded GPU encourages the use of many-fold data parallelism and thread parallelism, which readily scales to thousands of parallel threads on many processors. The GPU scalable parallel programming model for graphics and parallel computing is designed for transparent and portable scalability. A graphics program or CUDA program is written once and runs on a GPU with any number of processors. As shown in Section B.1, a CUDA programmer explicitly states both fine-grained and coarse-grained parallelism in a thread program by decomposing the problem into grids of thread blocks—the same program will run efficiently on GPUs or CPUs of any size in current and future generations as well.

Recent Developments

Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these programs run the application tens or hundreds of times faster than multicore CPUs are capable of running them. Examples include n-body simulation, molecular modeling, computational finance, and oil and gas exploration data processing. Although many of these use single-precision floating-point arithmetic, some problems require double precision. The recent arrival of double-precision floating-point in GPUs enables an even broader range of applications to benefit from GPU acceleration.

For a comprehensive list and examples of current developments in applications that are accelerated by GPUs, visit CUDAZone: www.nvidia.com/CUDA.

Future Trends

Naturally, the number of processor cores will continue to increase in proportion to increases in available transistors as silicon processes improve. In addition, GPUs will continue to enjoy vigorous architectural evolution. Despite their demonstrated high performance on data-parallel applications, GPU core processors are still of relatively simple design. More aggressive techniques will be introduced with each successive architecture to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is a new field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations.

Further Reading

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. SIGGRAPH 1988* (August), 239–46.

Akeley, K. [1993]. "RealityEngine Graphics." *Proc. SIGGRAPH 1993* (August), 109–16.

Blelloch, G. B. [1990]. "Prefix Sums and Their Applications". In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Francisco.

Blythe, D. [2006]. "The Direct3D 10 System", *ACM Trans. Graphics* Vol. 25, no. 3 (July), 724–34.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahlian, M. Houston, and P. Hanrahan [2004]. "Brook for GPUs: Stream Computing on Graphics Hardware." *Proc. SIGGRAPH 2004*, 777–86, August. http://doi.acm.org/10.1145/1186562.1015800.

Elder, G. [2002] "Radeon 9700." Eurographics/SIGGRAPH Workshop on Graphics Hardware, Hot3D Session, www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.

Fernando, R. and M. J. Kilgard [2003]. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, Reading, MA.

Fernando, R. (Ed.), [2004]. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley, Reading, MA. https://developer.nvidia.com/gpugems/GPUGems/gpugems_pref01.html.

Foley, J., A. van Dam, S. Feiner, and J. Hughes [1995]. *Computer Graphics: Principles and Practice, second edition in C*, Addison-Wesley, Reading, MA.

Hillis, W. D. and G. L. Steele [1986]. "Data parallel algorithms." *Commun. ACM* 29, 12 (Dec.), 1170–83. http://doi.acm.org/10.1145/7902.7903.

IEEE Std 754-2008 [2008]. *IEEE Standard for Floating-Point Arithmetic*. ISBN 978-0-7381-5752-8, STD95802, http://ieeexplore.ieee.org/servlet/opac?punumber=4610933 (Aug. 29).

Industrial Light and Magic [2003]. *OpenEXR*, www.openexr.com.

Intel Corporation [2007]. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November. Order Number: 248966-016. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

Kessenich, J. [2006]. *The OpenGL Shading Language, Language Version 1.20, Sept. 2006*. www.opengl.org/documentation/specs/.

Kirk, D. and D. Voorhies [1990]. "The Rendering Architecture of the DN10000VS." *Proc. SIGGRAPH 1990* (August), 299–307.

Lindholm E., M.J. Kilgard, and H. Moreton [2001]. "A User- Programmable Vertex Engine." *Proc. SIGGRAPH 2001* (August), 149–58.

Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym [2008]. "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro* Vol. 28, no. 2 (March–April), 39–55.

Microsoft Corporation. Microsoft DirectX Specification, https://msdn.microsoft.com/en-us/library/windows/apps/hh452744.aspx.

Microsoft Corporation [2003]. Microsoft DirectX 9 Programmable Graphics Pipeline, Microsoft Press, Redmond, WA.

Montrym, J., D. Baum, D. Dignam, and C. Migdal [1997]. "InfiniteReality: A Real-Time Graphics System." *Proc. SIGGRAPH 1997* (August), 293–301.

Montrym, J. and H. Moreton [2005]. "The GeForce 6800", *IEEE Micro*, Vol. 25, no. 2 (March–April), 41–51.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits", *Electronics*, Vol. 38, no. 8 (April 19).

Nguyen, H. (Ed.), [2008]. *GPU Gems 3*, Addison-Wesley, Reading, MA.

Nickolls, J., I. Buck, M. Garland, and K. Skadron [2008]. "Scalable Parallel Programming with CUDA", *ACM Queue* Vol. 6, no. 2 (March–April) 40–53.

NVIDIA [2007]. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

NVIDIA [2007]. *CUDA Programming Guide 1.1*. https://developer.nvidia.com/nvidia-gpu-programming-guide.

NVIDIA [2007]. *PTX: Parallel Thread Execution ISA version 1.1*. www.nvidia.com/object/io_1195170102263.html.

Nyland, L., M. Harris, and J. Prins [2007]. "Fast N-Body Simulation with CUDA." In H. Nguyen (Ed.), *GPU Gems 3,* Addison-Wesley, Reading, MA.

Oberman, S. F. and M. Y. Siu [2005]. "A High-Performance Area- Efficient Multifunction Interpolator," *Proc. Seventeenth IEEE Symp. Computer Arithmetic*, 272–79.

Patterson, D. A. and J. L. Hennessy [2004]. *Computer Organization and Design: The Hardware/Software Inter face*, third edition, Morgan Kaufmann Publishers, San Francisco.

Pharr, M. ed. [2005]. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA.

Satish, N., M. Harris, and M. Garland [2008]. "Designing Efficient Sorting Algorithms for Manycore GPUs," NVIDIA Technical Report NVR-2008-001.

Segal, M. and K. Akeley [2006]. *The OpenGL Graphics System: A Specification, Version 2.1, Dec. 1, 2006*. www.opengl.org/documentation/specs/.

Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens [2007]. "Scan Primitives for GPU Computing." In *Proc. of Graphics Hardware 2007* (August), 97–106.

Volkov, V. and J. Demmel [2008]. "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Technical Report No. UCB/EECS-2008-49, 1–11. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," In *Proc. Supercomputing* 2007, November.