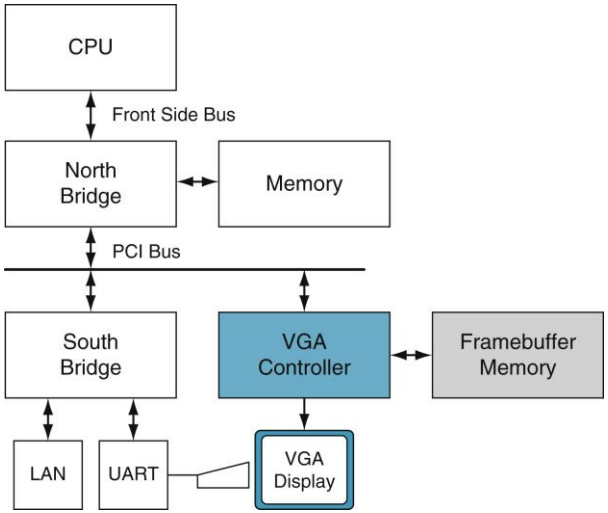



**FIGURE B.2.1 Historical PC.** VGA controller drives graphics display from framebuffer memory.

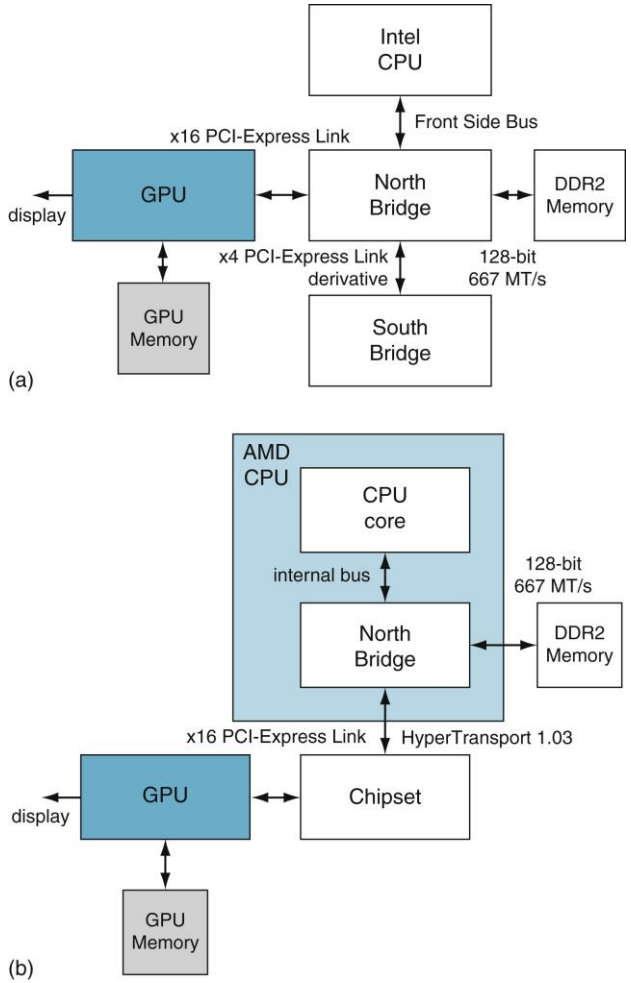



**FIGURE B.2.2 Contemporary PCs with Intel and AMD CPUs.** See Chapter 6 for an explanation of the components and interconnects in this figure.

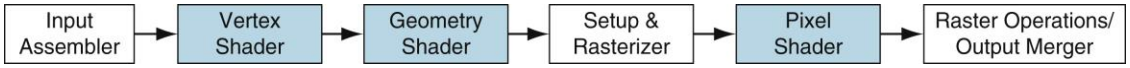



**FIGURE B.2.3 Graphics logical pipeline.** Programmable graphics shader stages are blue, and fixed-function blocks are white.

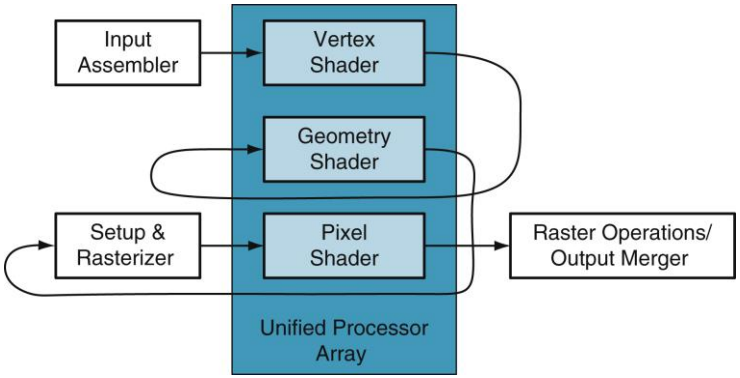



**FIGURE B.2.4 Logical pipeline mapped to physical processors.** The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

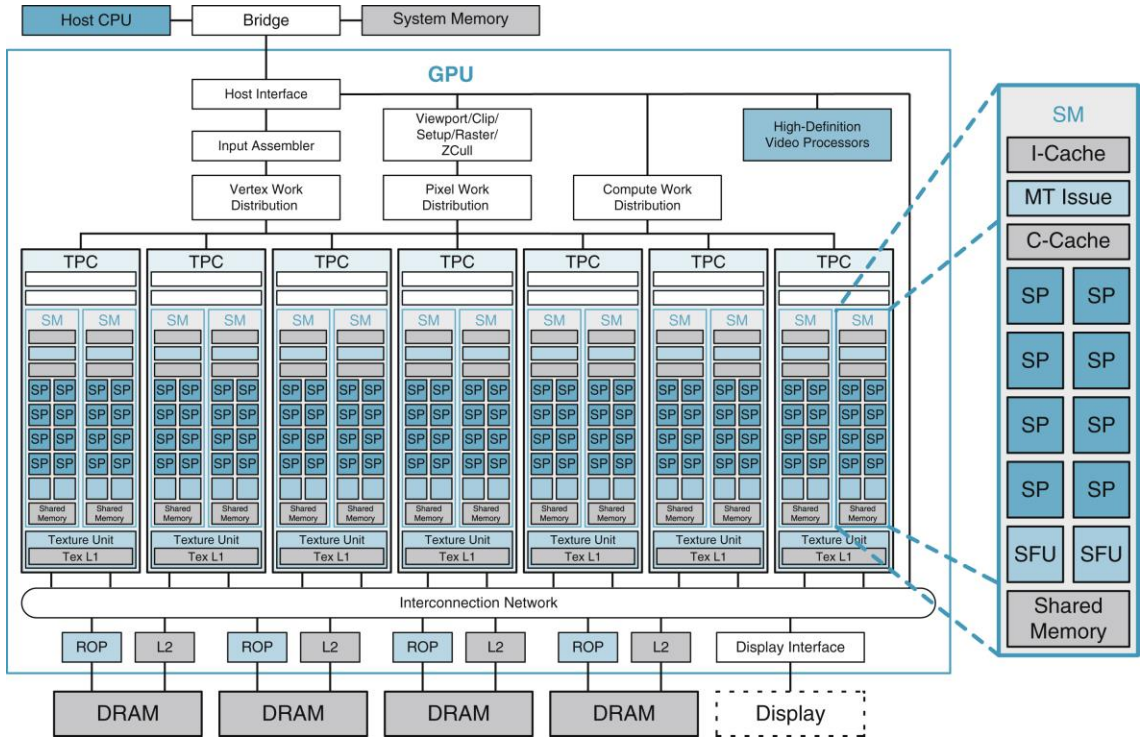



**FIGURE B.2.5 Basic unified GPU architecture.** Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM 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.

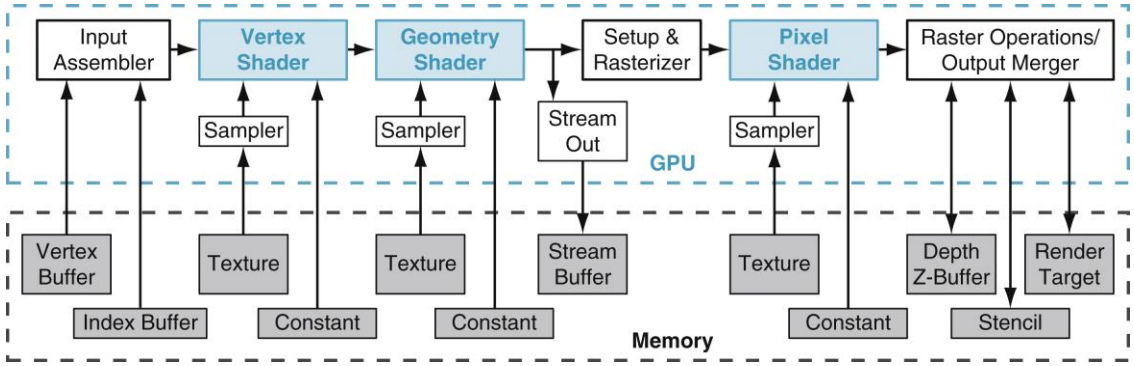



**FIGURE B.3.1 Direct3D 10 graphics pipeline.** Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are gray. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion.

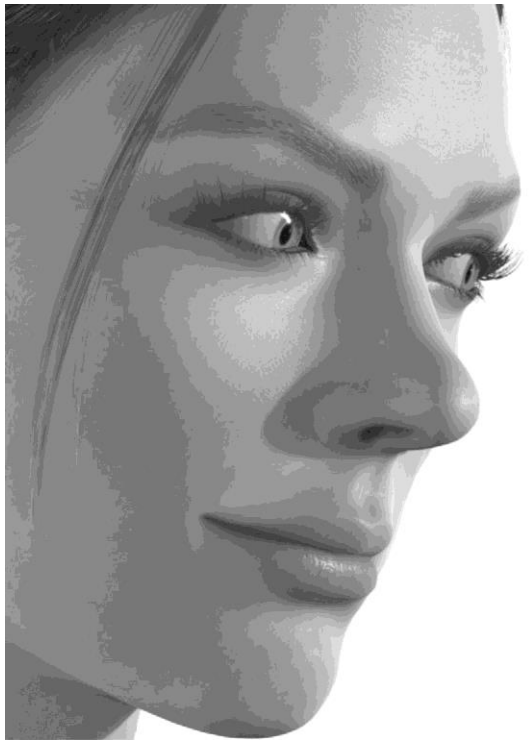


**FIGURE B.3.2 GPU-rendered image.** To give the skin visual depth and translucency, the pixel shader program models three separate skin layers, each with unique subsurface scattering behavior. It executes 1400 instructions to render the red, green, blue, and alpha color components of each skin pixel fragment.

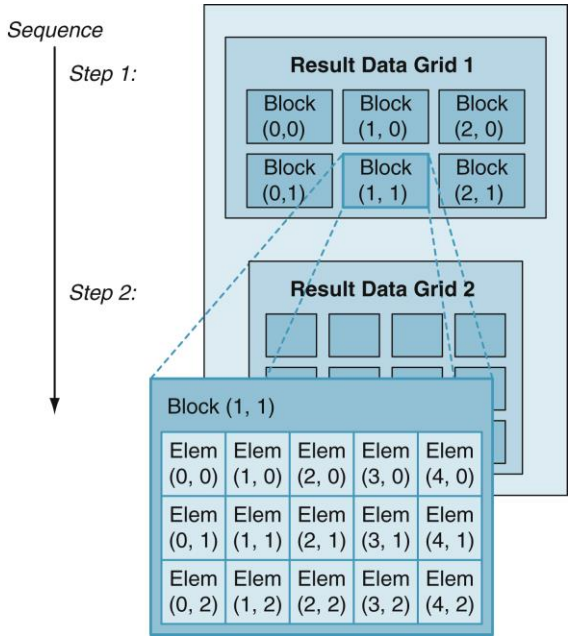



**FIGURE B.3.3 Decomposing result data into a grid of blocks of elements to be computed in parallel.**

***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.

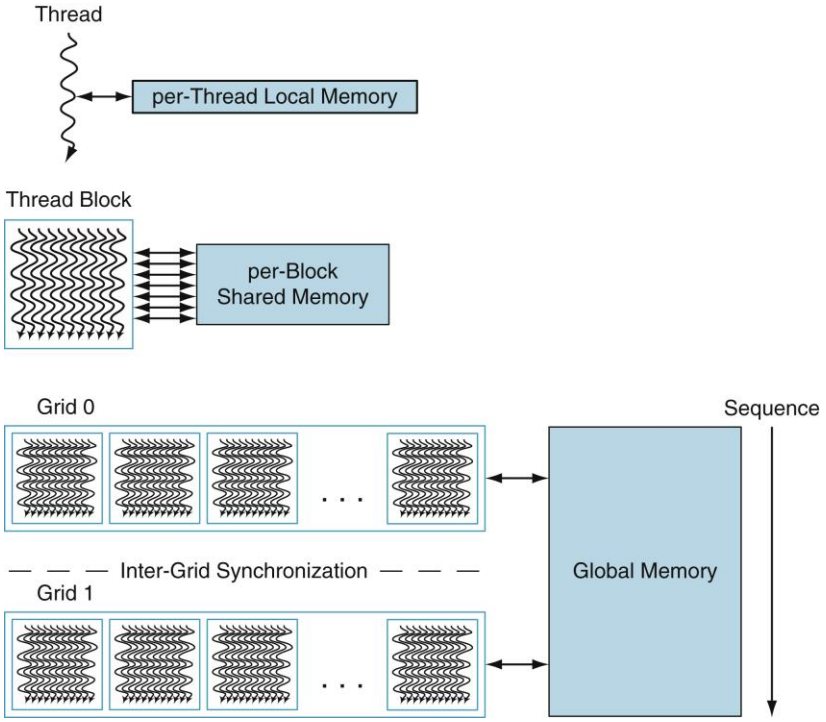



**FIGURE B.3.5 Nested granularity levels—thread, thread block, and grid—have corresponding memory sharing levels—local, shared, and global.** Per-thread local memory is private to the thread. Per-block shared memory is shared by all threads of the block. Per-application global memory is shared by all threads.

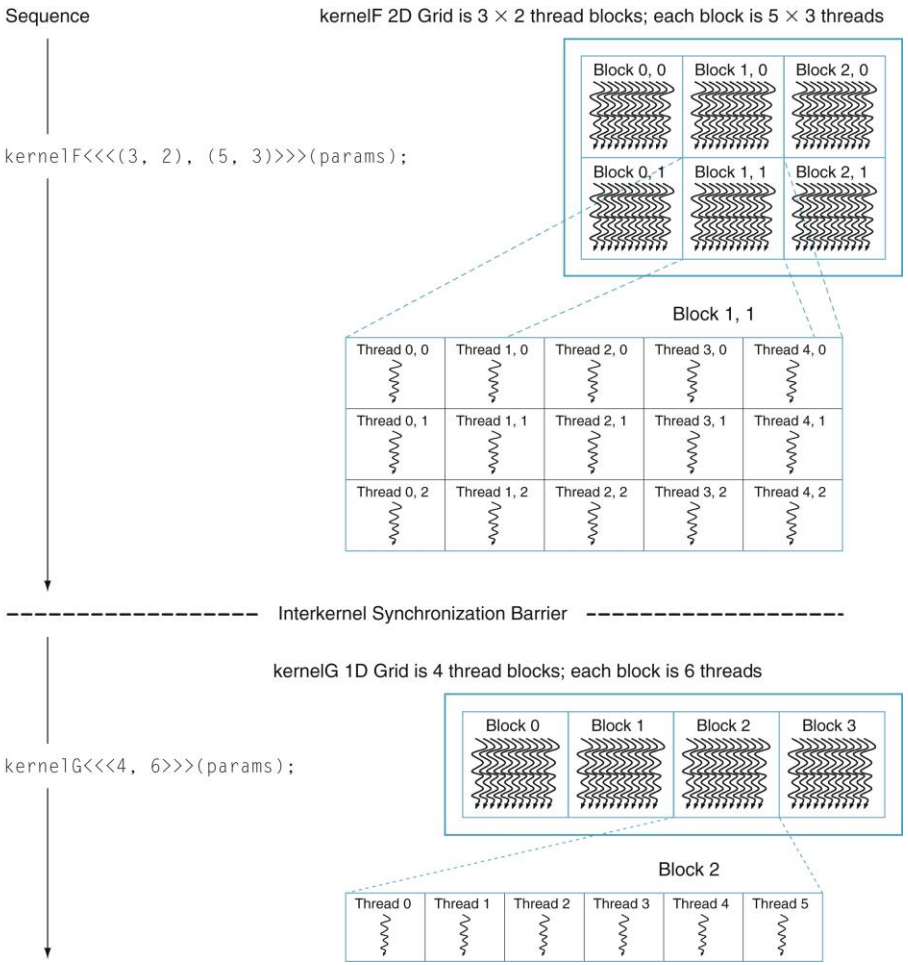



**FIGURE B.3.6 Sequence of kernel** *F* instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel *G* on a 1D grid of 1D thread blocks.

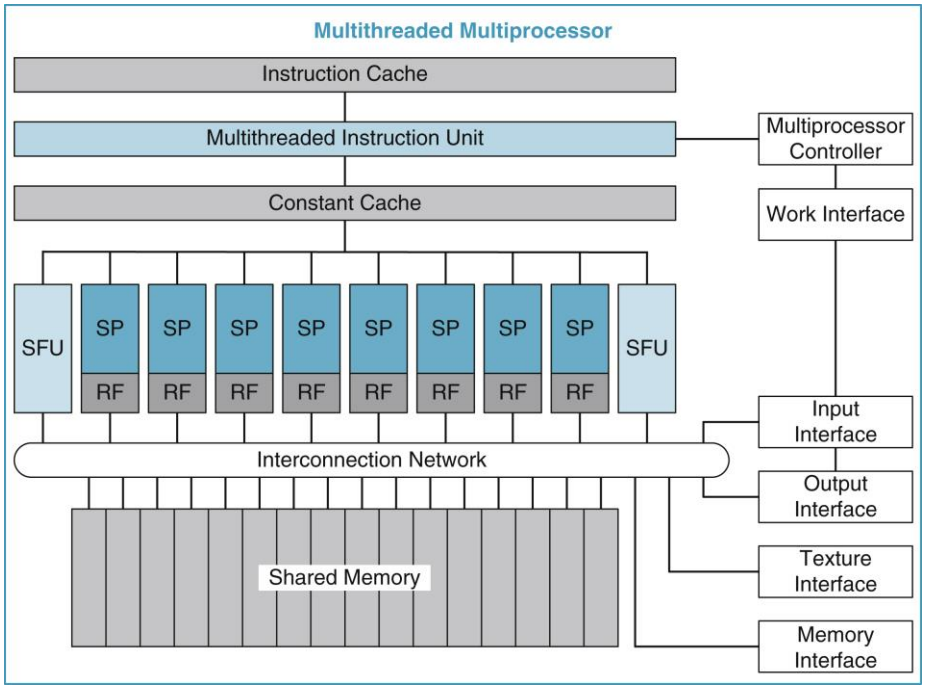



**FIGURE B.4.1 Multithreaded multiprocessor with eight scalar processor (SP) cores.** The eight SP cores each have a large multithreaded *register file* (RF) and share an instruction cache, multithreaded instruction issue unit, constant cache, two *special function units* (SFUs), interconnection network, and a multibank shared memory.

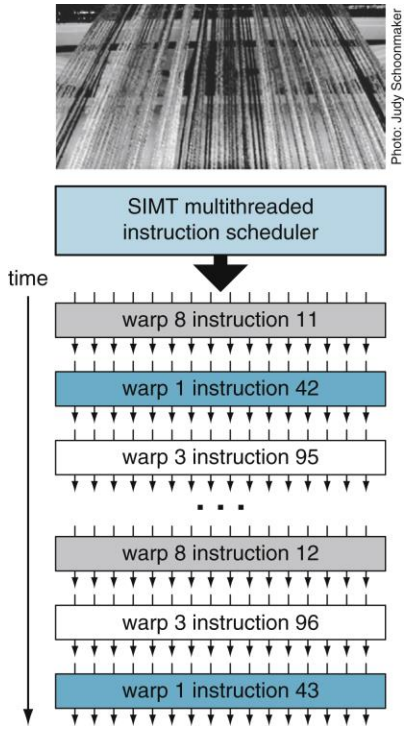



**FIGURE B.4.2 SIMT multithreaded warp scheduling.** The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time.

**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<br>atom.global.cas.b32 d,[a], b, c | atomic { d = *a;<br>*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.**

| 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).

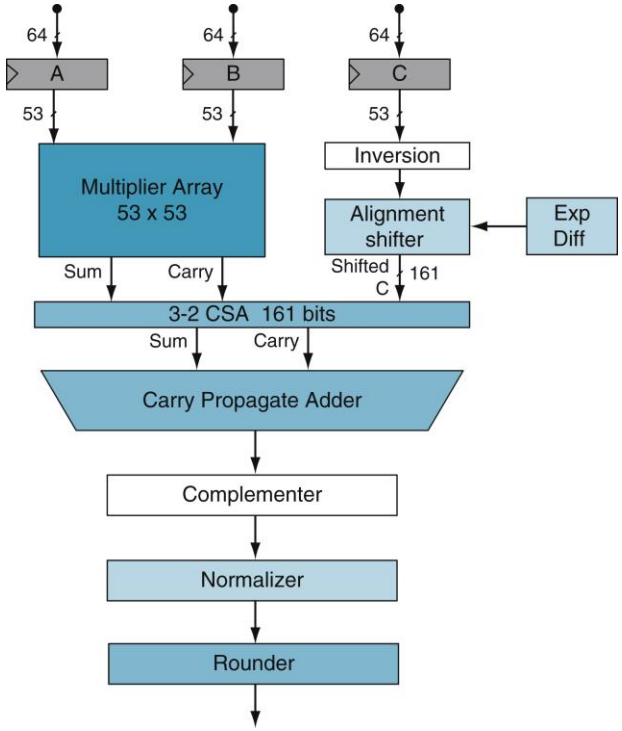



**FIGURE B.6.2 Double-precision fused-multiply-add (FMA) unit.** Hardware to implement floating-point A× B+ C for double precision.

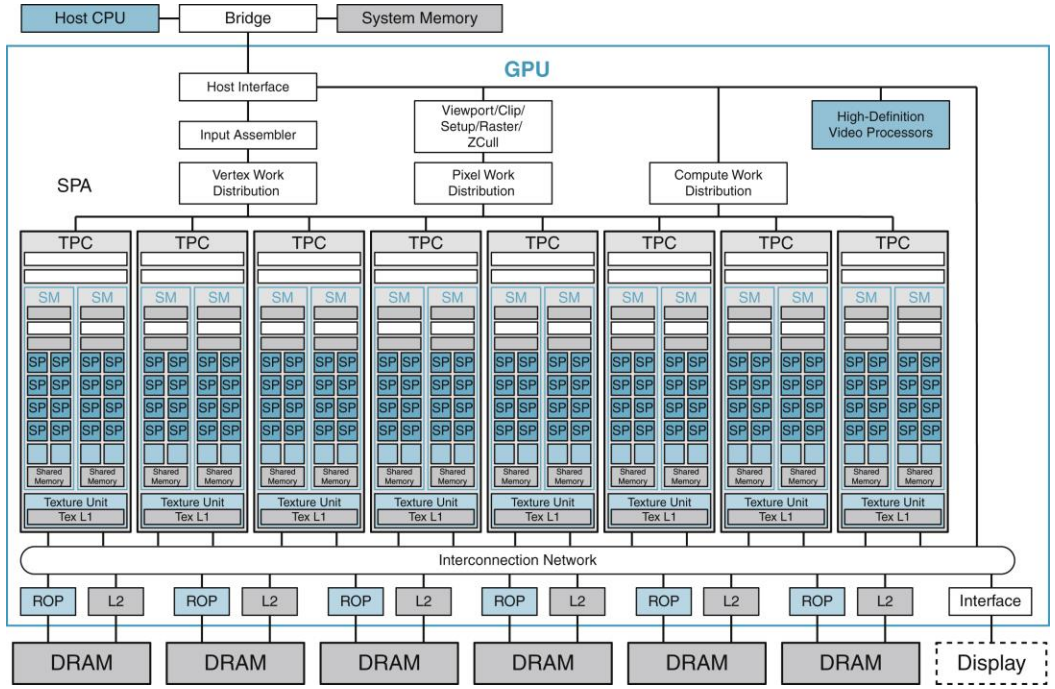



**FIGURE B.7.1 NVIDIA Tesla unified graphics and computing GPU architecture.** This GeForce 8800 has 12 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

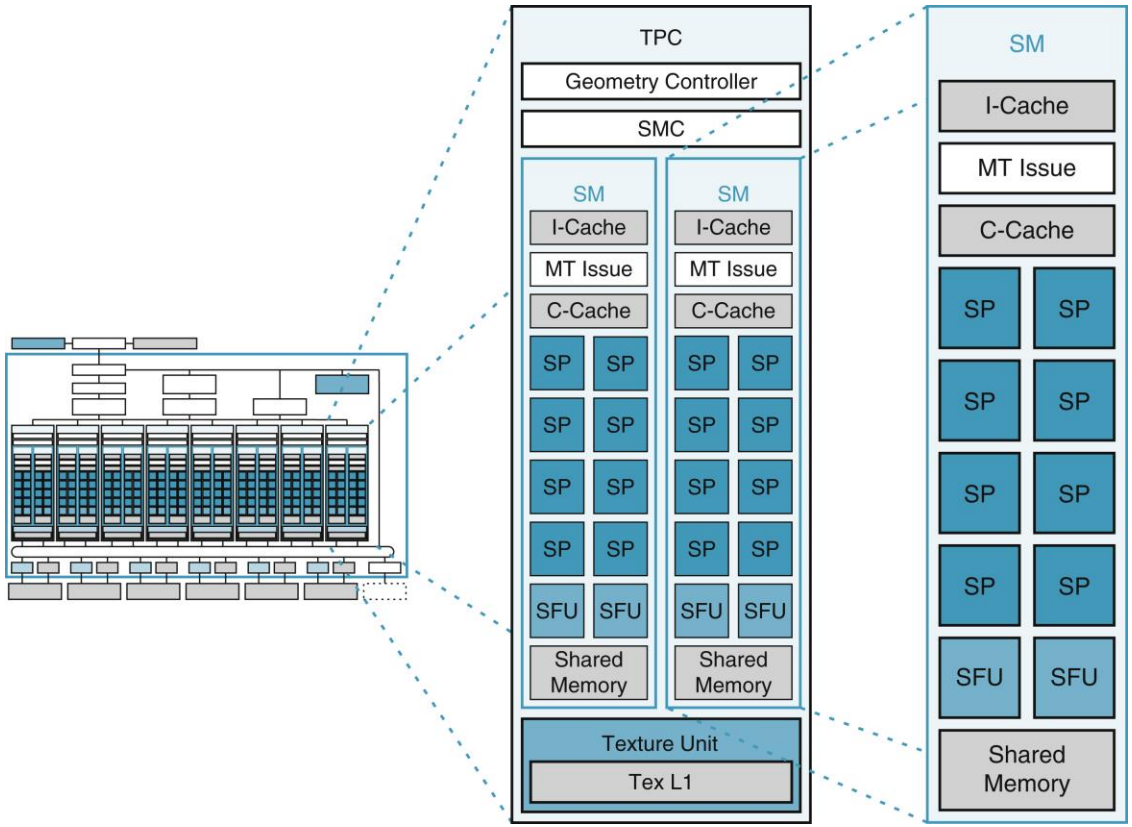



**FIGURE B.7.2 Texture/processor cluster (TPC) and a streaming multiprocessor (SM).** Each SM has eight *streaming processor* (SP) cores, two SFUs, and a shared memory.

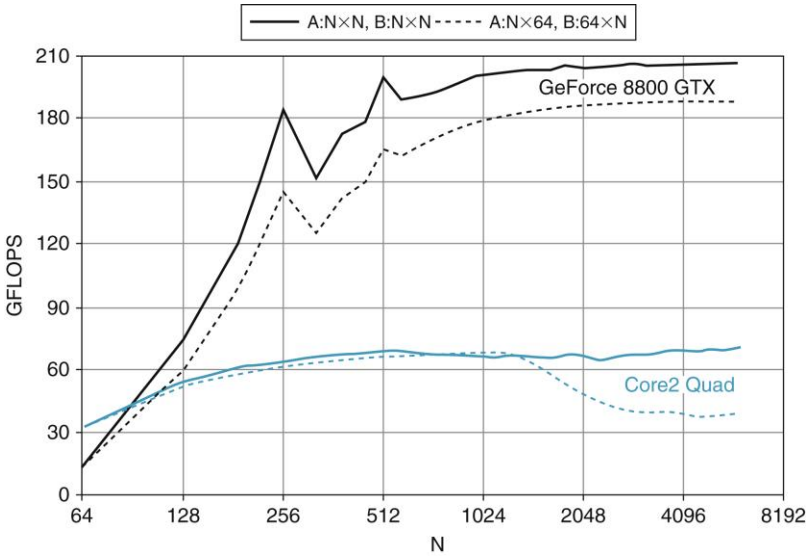



**FIGURE B.7.3 SGEMM dense matrix-matrix multiplication performance rates.** The graph shows single-precision GFLOPS rates achieved in multiplying square N×N matrices (solid lines) and thin N ×64 and 64×N matrices (dashed lines). Adapted from Figure 6 of Volkov and Demmel [2008]. The black lines are 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.

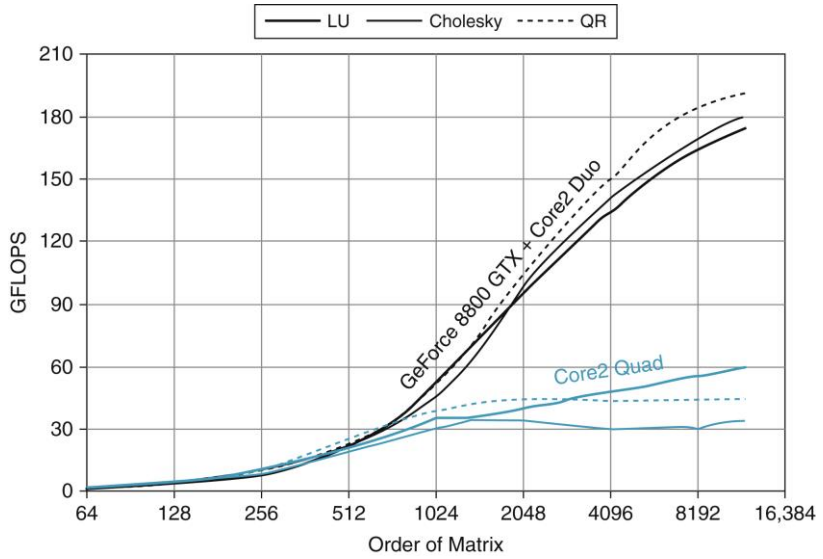



**FIGURE B.7.4 Dense matrix factorization performance rates.** The graph shows GFLOPS rates achieved in matrix factorizations using the GPU and using the CPU alone. Adapted from Figure 7 of Volkov and Demmel [2008]. The black lines are for a 1.35 GHz NVIDIA GeForce 8800 GTX, CUDA 1.1, Windows XP attached 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.

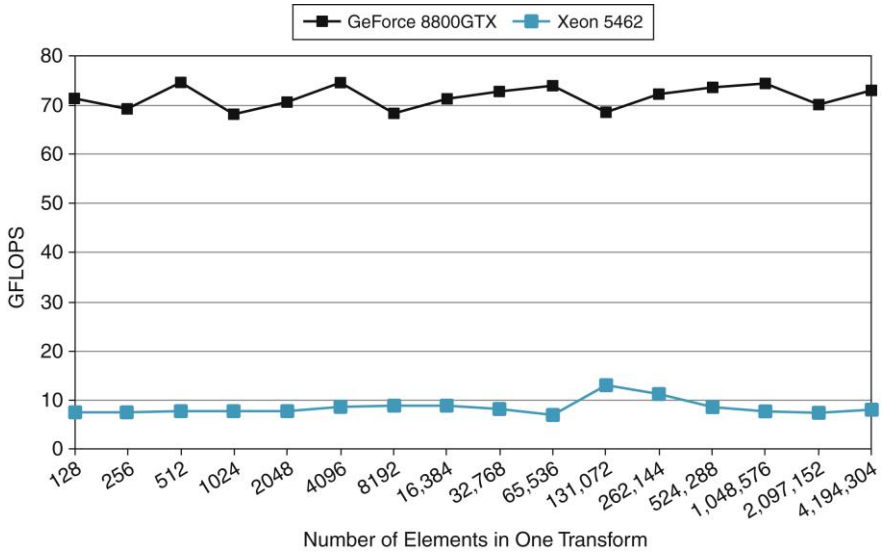



**FIGURE B.7.5 Fast Fourier transform throughput performance.** The graph compares the performance of batched one-dimensional in-place complex FFTs on a 1.35 GHz GeForce 8800 GTX with a quad-core 2.8 GHz Intel Xeon E5462 series (code named “Harpertown”), 6MB L2 Cache, 4GB Memory, 1600 FSB, Red Hat Linux, Intel MKL 10.0.

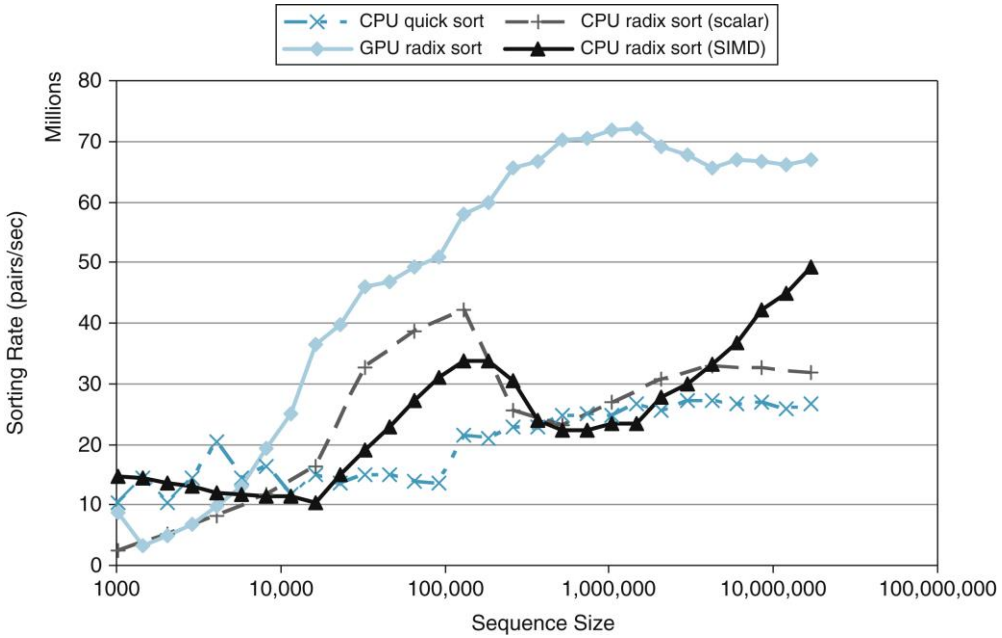
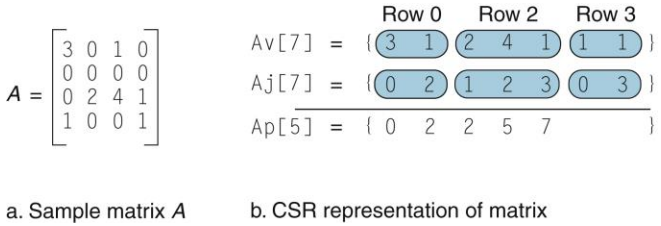



**FIGURE B.7.6 Parallel sorting performance.** This graph compares sorting rates for parallel radix sort implementations on a 1.5 GHz GeForce 8800 Ultra and an 8-core 2.33 GHz Intel Core2 Xeon E5345 system.




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.**

```
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.**

```
__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.**

```
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.**

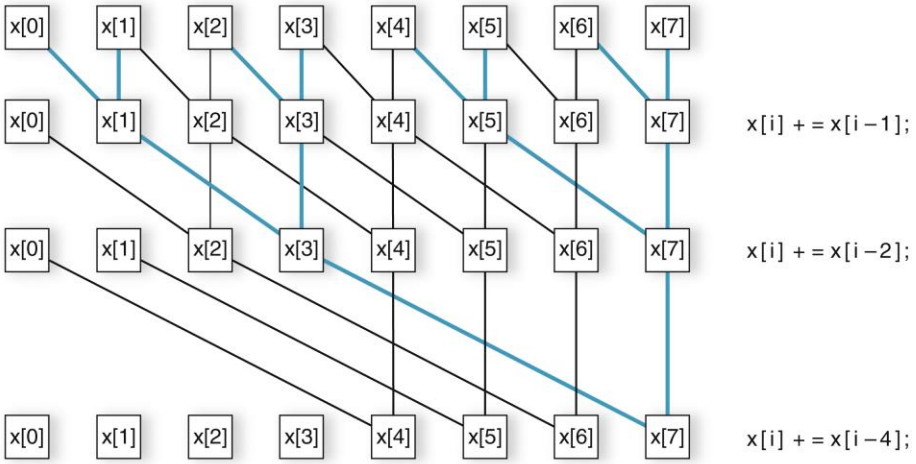



**FIGURE B.8.8 Tree-based parallel scan data references.**

```
__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.**

```
__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.**

```
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.**

```
__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.**

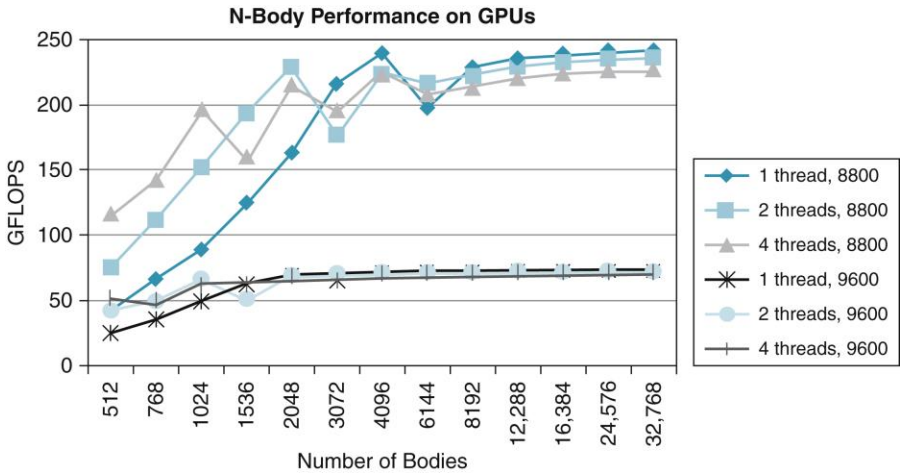



**FIGURE B.8.15 Performance measurements of the N-body application on a GeForce 8800 GTX and a GeForce 9600.** The 8800 has 128 stream processors at 1.35 GHz, while the 9600 has 64 at 0.80 GHz (about 30% of the 8800). The peak performance is 242 GFLOPS. For a GPU with more processors, the problem needs 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.

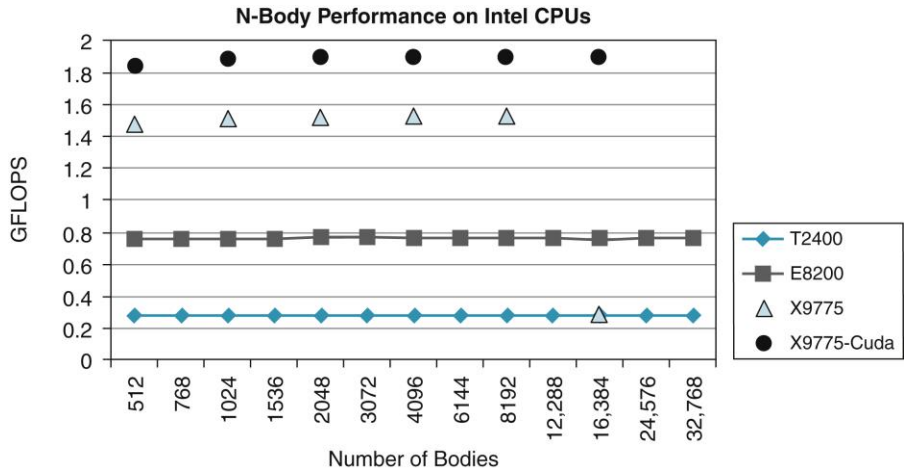



**FIGURE B.8.16 Performance measurements on the N-body code on a CPU.** The graph showssingle precision N-body performance using Intel Core2 CPUs, denoted by their CPU model number.Note the dramatic reduction in GFLOPS performance (shown in GFLOPS on the *y*-axis), demonstrating how much faster the GPU 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.

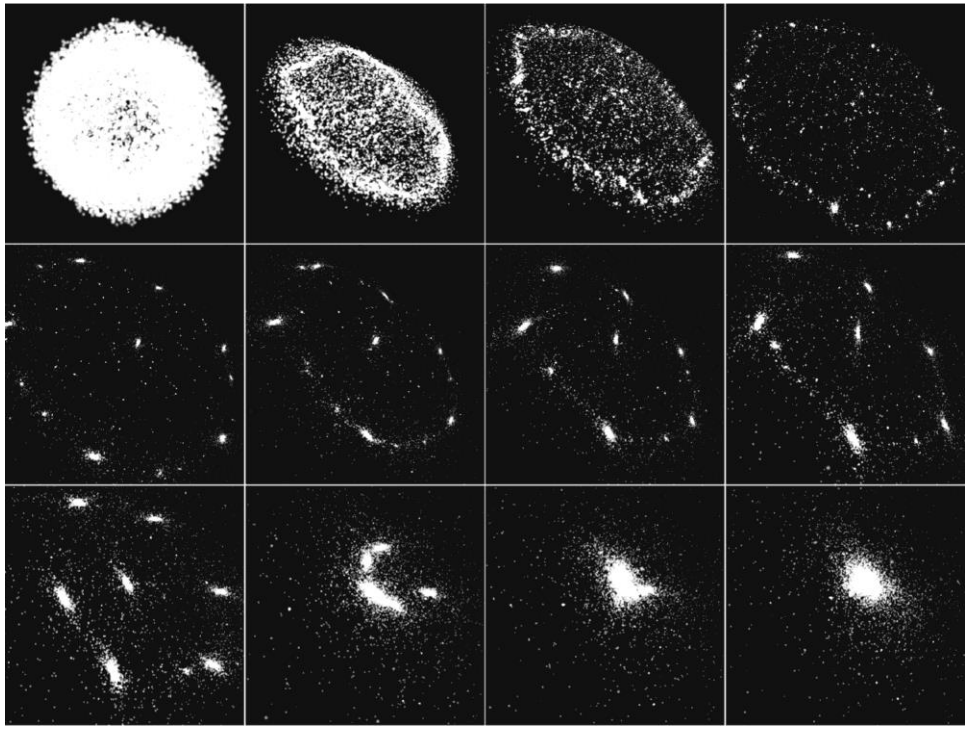


**FIGURE B.8.17 Twelve images captured during the evolution of an N-body system with 16,384 bodies.**

| 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 |