8.8 Real stuff: Mapping applications to GPUs

(Original section[1])

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.

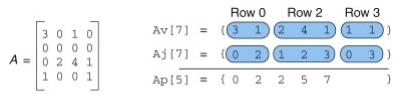
Sparse matrices

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. The figure below shows an example of the CSR representation of a simple matrix.

Figure 8.8.1: Compressed sparse row (CSR) matrix (COD Figure B.8.1).



a. Sample matrix *A* b. CSR representation of matrix

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 the figure below. 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 COD Figure B.8.3 (Serial code for sparse matrix-vector multiply).

Figure 8.8.2: Serial C code for a single row of sparse matrix-vector multiply (COD Figure B.8.2).

```
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 8.8.3: Serial code for sparse matrix-vector multiply (COD Figure B.8.3).

```
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);
    }
}
```

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 the figure below. 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).

Figure 8.8.4: CUDA version of sparse matrix-vector multiply (COD Figure B.8.4).

```
__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);
    }
}
```

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. Modifications using shared memory are shown in the figure below.

Figure 8.8.5: Shared memory version of sparse matrix-vector multiply (COD Figure B.8.5).

```
__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;
    }
```

```
}
```

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 the figure above. 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.

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:

$$\mathbf{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$$

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:

$$\mathbf{scan}(a, +) = [3\ 4\ 11\ 11\ 15\ 16\ 22\ 25]$$

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 the figure below.

Figure 8.8.6: Template for serial plus-scan (COD Figure B.8.6).

```
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];
}
```

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 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 the figure below. 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.

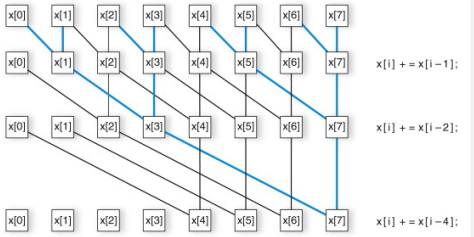
Figure 8.8.7: CUDA template for parallel plus-scan (COD Figure B.8.7).

```
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];
    }
```

To understand the action of this loop, consider the figure below, 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 8.8.8: Tree-based parallel scan data references (COD Figure B.8.8).



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.

The figure below 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.

Figure 8.8.9: CUDA implementation of plus-reduction (COD Figure B.8.9).

```
__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]);
}
```

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 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 the figure below 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.

Figure 8.8.10: CUDA code for radix sort (COD Figure B.8.10).

```
__device__ void radix_sort(unsigned int *values)
{
    for(int bit = 0; bit < 32; ++bit)
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}
```

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 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 the figure below.

Figure 8.8.11: CUDA code to partition data on a bit-by-bit basis, as part of radix sort (COD Figure B.8.11).

```
__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;
    }
```

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[2]

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:

$$f_{ij} = G\frac{m_i m_j}{\|r_{ij}\|^2} \times \frac{r_{ij}}{\|r_{ij}\|}$$

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 $O(n)$.

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 the figure below and COD Figure B.8.13 (CUDA thread code to compute the total force on a single body). 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.

Figure 8.8.12: Serial code to compute all pair-wise forces on N bodies (COD Figure B.8.12).

```
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 8.8.13: CUDA thread code to compute the total force on a single body (COD Figure B.8.13).

```
__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;
}
```

The outer loop is replaced by a CUDA kernel grid that launches *N* threads, one for each 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 the figure below.

Figure 8.8.14: CUDA code to compute the total force on each body, using shared memory to improve performance (COD 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]);
        }
```

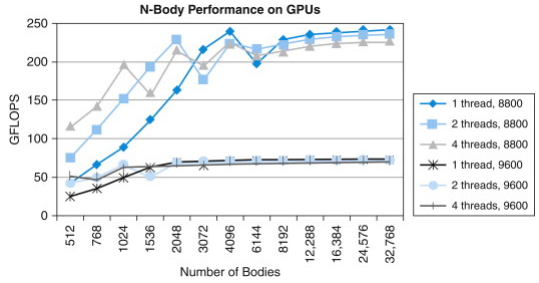
```
            __syncthreads();
        }
        accel[i] = acc;
}
```

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 the figure below.

Figure 8.8.15: Performance measurements of the N-body application on a GeForce 8800 GTX and a GeForce 9600 (COD Figure B.8.15).

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.



Using multiple threads per body

The figure above 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*.

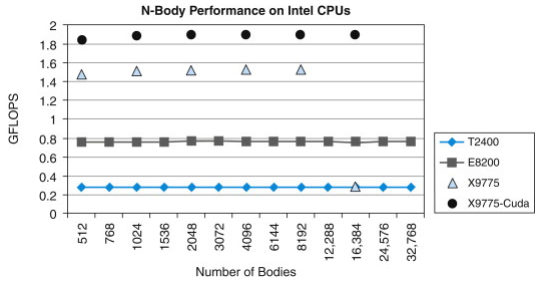
As an example, the performance on the 8800 GTX jumps by 110% when *N* = 1024 (one thread achieves 90 GFLOPS, where four achieve 190 GFLOPS). Performance degrades slightly on large *N*, so we only use this optimization for *N* smaller than 4096. The performance increases are shown in the figure above 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 the figure above and the figure below. In the figure above, 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 8.8.16: Performance measurements on the N-body code on a CPU (COD Figure B.8.16).

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 figure above 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.

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).

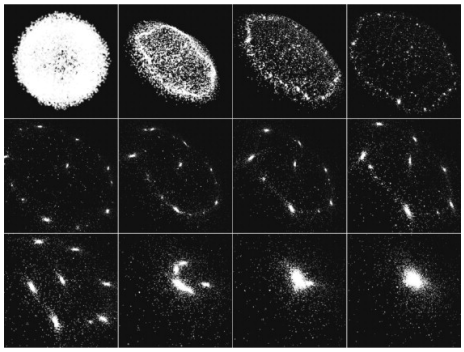
How can the GPU speed up the code by such a large amount? The answer requires inspecting architectural details. The pair-wise force calculation requires 20 floating-point operations, comprised mostly of addition and multiplication instructions (some of which can be combined using a multiply-add instruction), 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,[3] although this has improved in the latest Penryn CPU family with its faster 4-bit divider.[4] 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 COD Section B.6 (Floating-point arithmetic) 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.

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.

The figure below 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 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.

Figure 8.8.17: Twelve images captured during the evolution of an N-body system with 16,384 bodies (COD Figure B.8.17).



(*1) This section is in original form.

(*2) Adapted from Nyland et al. [2007], "Fast N-Body Simulation with CUDA," Chapter 31 of *GPU Gems 3*.

(*3) The x86 SSE instructions reciprocal-square-root (RSQRT*) and reciprocal (RCP*) were not considered, as their accuracy is too low to be comparable.

(*4) Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November 2007. Order Number: 248966-016. Also available at www3.intel.com/design/processor/manuals/248966.pdf.