Compute Unified Device Architecture Application Suitability

Graphics processing units (GPUs) can provide excellent speedups on some, but not all, general-purpose workloads. Using a set of computational GPU kernels as examples, the authors show how to adapt kernels to utilize the architectural features of a GeForce 8800 GPU and what finally limits the achievable performance.

Computational scientists have long been interested in graphics processing units (GPUs) due to their relatively low cost per unit of floating-point (FP) performance. Unlike conventional multiprocessors, a GPU's processor cores are specialized for program behaviors common to graphics shaders—thousands of independent threads, each comprising only dozens or hundreds of instructions, performing few memory accesses and producing a small number of output values.[1] Recent advances in hardware and programmability have opened GPUs to a broader community of developers. GPUs' throughput-optimized architectural features can outstrip CPU performance on numerical computational workloads, depending on how well the workload matches the computational behavior for which the GPU is designed.

An important question for many developers is whether they can map particular applications to these new GPUs to achieve significant performance increases over contemporary multicore processors. In this article, we describe our findings through an effort in mapping a wide variety of numerical applications to the Nvidia GeForce 8800 GTX using its compute unified device architecture (CUDA).

1521-9615/09/$25.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Wen-Mei Hwu, Christopher Rodrigues, Shane Ryoo, and John Stratton
University of Illinois, Urbana-Champaign

Programming for GPU Performance

Each GeForce-8 Series GPU is effectively a large set of processor cores with the ability to directly address a global memory. This allows for a more general and flexible programming model than previous generations of GPUs, making it easier for developers to implement data-parallel kernels of numerical applications. CUDA and GeForce 8800 have specific microarchitectural features that appeal to application developers—the CUDA programming guide has a more complete description (www.developer.nvidia.com/object/cuda.html).

Threading Model

In the CUDA programming model, the system consists of a host that's a traditional CPU and one or more massively data-parallel coprocessing compute devices. A CUDA program consists of multiple phases that are executed on either the host or a compute device such as a GPU (Figure 1). The program designates the phases that exhibit little or no data parallelism in host (CPU) code and compiles it with the host's standard C compiler,

which runs as an ordinary process. The phases that exhibit rich data parallelism are implemented in the device (GPU) code, which is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and associated data structures. Host code uses a CUDA-specific function-call syntax to invoke kernel code.

The runtime system executes kernels as batches of parallel threads in a single-program, multiple-data (SPMD)[2] programming style, in which kernels specify all simultaneous threads. These kernels typically comprise thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a fine-grained decomposition of work; for example, the kernel might compute each result array's element in a separate thread.

We can use dense matrix multiplication to illustrate the CUDA threading model. In this example, each thread calculates one product-matrix element, which involves a dot product of a row of the first input array and a column of the second input array, shown as the kernel function `matrixMul()` in Figure 2b. The kernel takes a pointer to a row of the first input matrix **A** and a pointer to a column of the second input matrix **B**, performs a dot product, and writes the value into an element of the output array **C**.

CUDA's threads are organized into a two-level hierarchy, at the highest of which all threads in a data-parallel execution phase form a grid. Figure 1 shows an example of thread organization and calls to two different kernel functions. Each call to a kernel initiates a thread grid; the system normally waits for all threads in the grid to complete before it lets the next grid begin. Each grid consists of many thread groupings, called thread blocks. All blocks in a grid have the same number of threads, with a maximum of 512. In Figure 1, the calls to kernels 1 and 2 create grids, wherein each thread block has 192 and 256 threads, respectively. Each thread in a thread block has a unique ID in the form of a three-dimensional (3D) coordinate, and each block in a grid also has a unique two-dimensional (2D) coordinate. Threads determine the work that they must do and the data they'll access by inspecting their own thread and block IDs.

Threads in a thread block can also communicate and barrier-synchronize with one another. This coordination feature distinguishes CUDA's programming model from shader programming of previous general-purpose computing on GPU (GPGPU) models. Coordination is essential for some parallel algorithms and many optimizations that boost performance.[3]

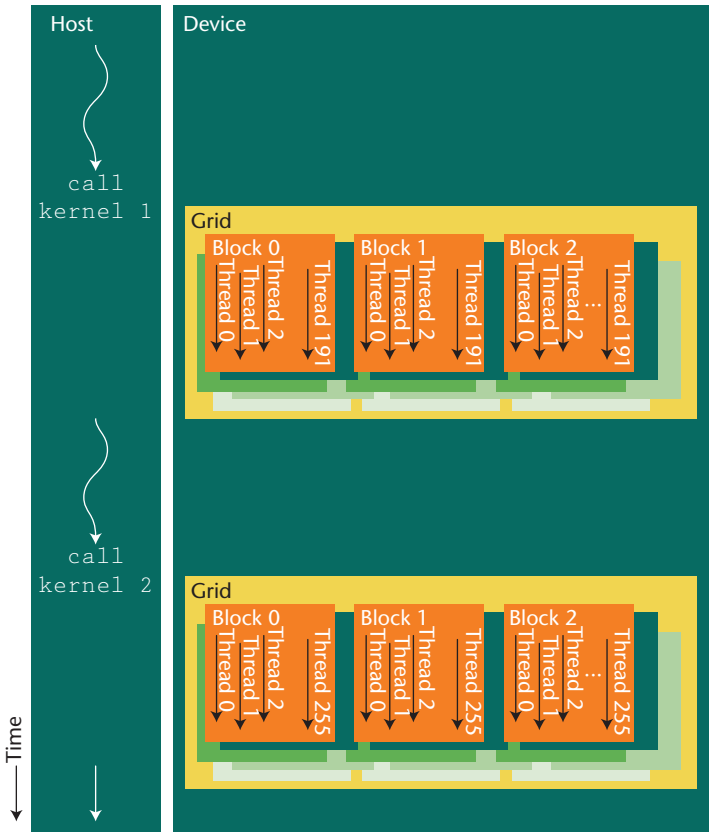

Figure 1. Compute unified device architecture (CUDA) threading structure. Threads are the fundamental units of parallel execution in CUDA. Each call to a CUDA kernel function creates a grid, which is a two-level hierarchy of threads. The hardware maintains thread IDs so that threads can manage themselves in aspects such as what part of the data structure to process.

Figure 2 shows a matrix multiplication example that multiplies two 4,096 × 4,096 element matrices. In this kernel, each thread block comprises 256 threads organized into two dimensions of 16 threads in each dimension. Each thread block calculates one 16 × 16 product matrix's submatrix. The kernel generates 256 × 256 thread blocks to cover the entire output matrix. The thread and thread-block organization is set in host code.

When host code invokes a kernel, it sets the grid and thread-block dimensions by passing them as parameters. Figure 2a shows two declared structures of type `dim3`: the first defines the configuration of blocks as 16 × 16 groups of threads in our example. The second is for the grid, which consists of 256 × 256 blocks. The final line of code invokes the kernel (see Figure 2b). Each thread calculates the starting positions in the input matrices based on its unique block and thread coor-

```
    float * gpu_A, float * gpu_B, float * gpu_C;

    // allocate GPU input and output matrices
    cudaMalloc((void**) &gpu_A, mem_sizeA);
    cudaMalloc((void**) &gpu_B, mem_sizeB);
    cudaMalloc((void**) &gpu_C, mem_sizeC);

    // copy input matrices from host to device
    cudaMemcpy{gpu_A, host_A, mem_size_A,
              cudaMemcpyHostToDevice);
    cudaMemcpy{gpu_B, host_B, mem_size_B,
              cudaMemcpyHostToDevice) );

    // set up execution parameters
    dim3 blocks(16, 16);
    dim3 grid(256, 256);

    // execute the kernel
(a) matrixMul<<< grid, blocks >>>(gpu_C, gpu_A, gpu_B);


    _global_ void
    matrixMul( float* C, float* A, float* B)
    {
      // Calculate index of the first element of A
      int indexA = 16 * blockIdx.y + threadIdx.y;
      ...

      // Initialize the result to 0
      float Csub = 0;

      for (i = 0; i < widthA; i++)
        {
          Ctemp += A[indexA] * B[indexB];
          indexA++;
          indexB += widthB;
        }
      C[c] = Csub;
(b) }
```

Figure 2. Matrix multiplication example. (a) The host code sets up and executes the kernel, and (b) the kernel code shows the workings of a kernel function and how it can be invoked with a particular thread configuration.

dinates. It then iterates through a loop to calculate the result and finally stores it to memory.

The way the runtime system instantiates a CUDA kernel as SPMD threads with unique IDs is similar to the way a parallel loop executes in other languages, such as a `parallel for` loop in OpenMP.[4] Whereas OpenMP lets all threads access a single shared memory, CUDA exposes the multiple memory spaces of a GPU's memory system. The host, which has a separate memory from the device's, uses API calls to allocate memory on the GPU and transfer data between memories. Different memory spaces on the device are also separate from one another. The developer bears the responsibility of selecting the appropriate data placement and layout for a given application, which requires knowledge of each memory's characteristics, as we'll explain later.

Other programming languages that support GPGPUs include Brook[5] and Accelerator.[6] Similar to CUDA, Brook requires the developer to divide the program into kernels and manage GPU resources. A Brook application performs data-parallel operations by calling a kernel function on one or more streams, which are collections of values that can be distributed to or collected from parallel computations. A Brook programmer can use streams in data-parallel reductions, scatter operations (indexed writes), or gather operations (indexed reads). Unlike CUDA thread blocks, the Brook programming model doesn't permit synchronization or communication between kernel iterations processing separate stream elements. Accelerator provides high-level data-parallel array operations through a library in the C# programming language. The accelerator runtime system dynamically and transparently compiles array operation sequences to GPU kernels, which doesn't expose any aspect of the GPU to the developer. Both Brook and Accelerator limit the size and complexity of GPU code due to their underlying graphics' API-based implementations.

Microarchitectural Support for Parallel Execution

The GeForce 8800 is a multiprocessor that's heavily specialized for graphics processing. Figure 3 depicts GeForce 8800's microarchitecture, which consists of 16 streaming multiprocessors (SMs), each containing eight streaming processors (SPs), or processor cores, running at 1.35 GHz. Cores in an SM execute instructions in single-instruction, multiple-data (SIMD) fashion, with the SM's instruction unit broadcasting the current instruction to the cores. Each core has one 32-bit, single-precision FP multiply-add arithmetic unit that can also perform 32-bit integer arithmetic. Additionally, each SM has two super functional units (SFUs) that execute more complex FP operations such as trigonometry functions with high throughput. The arithmetic units and the SFUs are fully pipelined, yielding 388.8 Gflops of peak theoretical performance for the GPU.

The GeForce 8800's unit of SIMD execution is a warp of 32 parallel threads. The CUDA runtime system forms warps from contiguous groups of threads in a thread block: the first 32 threads form the first warp, and so on. Although CUDA code doesn't explicitly declare warps, knowledge of them can enable useful code and data optimizations on the GeForce 8800. The hardware selects and executes one warp at a time. When threads within a warp take different control-flow paths (a

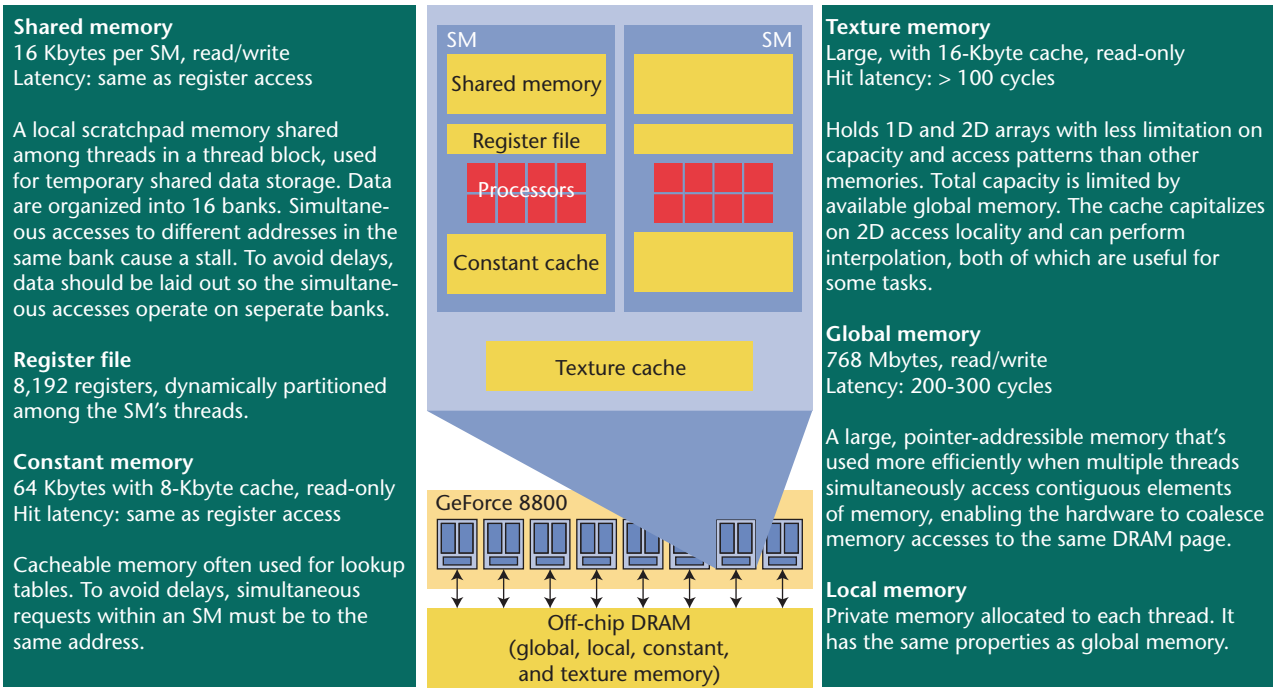

Figure 3. Processor organization and memory characteristics of the GeForce 8800. Code executing on a processor can access several memory spaces. Memories sited physically close to a processor (left column) are smaller and faster than more distant memories (right column).

situation known as branch divergence), the hardware executes multiple passes through the code with thread suppression on divergent paths to complete execution. Execution is slowed as much as if each thread had executed all control-flow paths involved. This effect makes kernels with a large number of data-dependent control flows unsuitable for the GPU.

The GPU generates SIMD execution from threads "on the fly," saving programmers the effort of manually restructuring control flow and data into SIMD form. This works well for graphics shaders and many data-parallel kernels in which all threads execute the same instruction sequences. In contrast, programmers must forecast control flow within a SIMD execution unit as predication for multicore architectures such as Larrabee.[7] The AMD/ATI runtime system transparently compiles shaders or Brook compute kernels to predicated SIMD code for AMD's FireStream.[8] The availability of SIMD scatter-gather memory operations and predication in AMD GPUs makes explicit SIMD nearly equivalent to CUDA's warp-based execution; the latter's main advantage is programming convenience.

An SM can perform zero-overhead scheduling to interleave warps and hide the latency of long-latency arithmetic and memory operations. When one warp stalls, the SM can instantly switch to a ready warp resident in the SM—the SM stalls only if no warps with ready operands are available. Scheduling freedom is high in many applications because threads in different warps are independent with the exception of explicit barrier synchronizations among threads in the same thread block. This lets the GPU maintain highly parallel execution using a small amount of chip area, in contrast to area-hungry speedup methods such as large caches or out-of-order execution.

Each SM supports a maximum of 768 simultaneously active thread contexts. The CUDA runtime assigns an integral number of up to eight thread blocks to an SM at any time to fill these thread contexts. When assigning a thread block to an SM, the CUDA runtime system automatically allocates the necessary amount of several hardware resources including thread contexts, shared memory, and registers. When optimizing device code, developers need to be aware of how these limits affect the number of parallel threads that can run on the device. Some conventional code optimizations might have negative effects in some cases because small increases in resource us-

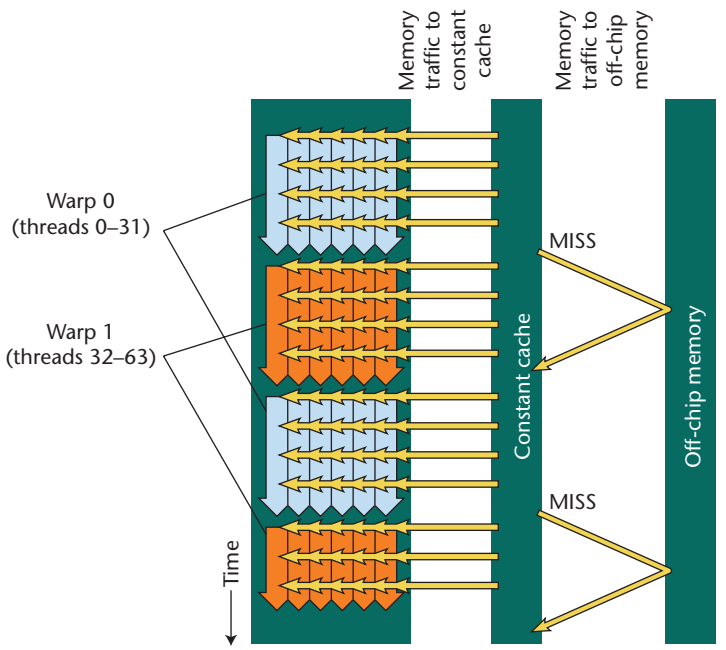

Figure 4. Constant memory. A fast, single-ported cache can feed many loads (gold arrowheads) with few cache accesses (gold arrows).

age can cause fewer thread blocks, and thus many fewer threads, to simultaneously execute.

Since the introduction of the GeForce 8800, other vendors have introduced similar architectures that combine graphics and general-purpose workloads. Like the GeForce 8800, AMD's FireStream[8] is a GPU with specialized memory systems and SIMD hardware. FireStream has scatter-gather operations, but it doesn't support pointers or have an equivalent to thread blocks. Currently, developers can choose between a high-level stream programming language, Brook, or a low-level assembly-like language, CAL (compute abstraction layer). FireStream and the more recent Nvidia Tesla GPUs support double-precision FP. The proposed Larrabee[8] graphics processor adheres to a multicore design philosophy and consists of single-threaded, in-order x86 cores with 16-wide SIMD units. The cores maintain shared-memory coherence through a ring network that connects their L2 caches. The hardware doesn't utilize multithreading to tolerate latency but provides explicit cache-control instructions that programmers can use to avoid waiting for anticipatable memory accesses.

Memory Spaces

Figure 3 describes the device's accessible memories. The developer can place data in global, shared, local, constant, or texture memory. The GPU's memories are specialized and have different access times and throughput limitations. Some memories furnish fast access only for limited patterns of memory references. Consequently, developers must use their understanding of the memory system to structure both data and kernel code for high performance. The layout of key data structures often determines kernel performance.

Global memory is a large, long-latency memory that exists physically as off-chip dynamic RAM (DRAM); it serves the same purpose as main memory in a chip multiprocessor. The developer must write a kernel's output to global memory to be readable after the kernel terminates. To avoid wasting hundreds of cycles while a thread waits for a long-latency global-memory load or store to complete, a common technique is to execute batches of global accesses, one per thread, exploiting the hardware's warp scheduling to overlap the threads' access latencies. Global-memory bandwidth is very high at 86.4 Gbyte/sec, but memory bandwidth can saturate if many threads request access within a short period of time. In addition, the system can sustain this bandwidth only when warps access contiguous 16-word lines; in other cases, the achievable bandwidth is a fraction of the maximum. Code transformations to coalesce accesses into 16-word lines and reuse data are generally necessary to achieve good performance. If a developer doesn't know a kernel's memory-access pattern in advance, it's typically not possible to overlap latencies or form contiguous accesses, which results in painfully slow access times to global memory.

Constant memory is specialized for situations in which many threads will read the same data simultaneously. A value read from the constant cache is broadcast to all threads in a warp, effectively serving 32 loads from memory with a single-cache access. This enables a fast, single-ported cache to feed multiple simultaneous memory accesses, as illustrated schematically in Figure 4. Recall that all threads in a warp execute the same instruction at any point in time. When all load-instruction instances access the same constant-memory location, the value at the location is broadcast to all threads, as shown by the horizontal arrows crossing the threads in a warp (see Figure 4). The GPU's tightly coupled cores let constant memory effectively multiply the GPU's memory bandwidth by 32, compared to a regular multiprocessor. The most dramatic performance gains we observed were in kernels that could take advantage of this effect.

Two magnetic resonance imaging (MRI)

benchmarks, abbreviated MRI-Q and MRI-FHD, demonstrate the use of constant memory. The raw digital data that an MRI scanner produces consists of many sample measurements in the frequency domain, which must be transformed into the spatial domain to produce an image. Fast algorithms such as the fast Fourier transform (FFT) are applicable, but advanced iterative algorithms can significantly increase the signal-to-noise ratio and decrease the artifacts in the reconstructed image. When using the advanced iterative algorithm described in another study,[9] computing the vector FHD is the most time-consuming step in the reconstruction. The FHD kernel transforms each sample into the spatial domain and sums the results for each coordinate $\vec{x}$ in a Cartesian lattice, where each point corresponds to a pixel in the reconstructed image. The pseudocode for FHD is as follows:

```
for each coordinate x⃗ in lattice, do
  a ← 0
  for each (k⃗, φ) in samples, do
      {Add the contribution of k⃗
         to the signal at x⃗ }
      a ← a + φ * e^(πi(k⃗·x⃗))
  fhd[x⃗] ← a
```

To map this pseudocode to CUDA, the developer assigns each outer loop's iteration to a separate thread. The kernel function contains the inner loop. The key indicator that constant memory suits this code is that all threads in the inner loop traverse the same sample data. As the kernel executes, the warp's threads will simultaneously load the same $\vec{k}$ and ϕ values from constant memory. Hardware detects that the addresses are equal and performs a single-cache access for all loads.

In vectorized code on a CPU, loading a scalar value from cache into all elements of a vector register achieves the effect of a constant-memory load. Using SSE (streaming SIMD extensions), the developer does this with a load instruction followed by a shuffle instruction. As we show later, GPU kernels relying on constant memory can outperform CPU code using SSE. This is largely due to the fact that SSE only exploits four-way SIMD parallelism per CPU core, whereas an entire GeForce 8800 is 128-wide.

Texture memory holds 1D- or 2D-array data and takes advantage of 2D access locality. Unlike other memories, its performance doesn't suffer under irregular, random-access patterns, but its access latency is quite long. A good example of texture memory's use is in the motion estimation stage of an H.264 video encoder. The H.264 kernel that we evaluated accelerates a full-search motion estimation algorithm. The kernel compares small 4 × 4 pixel blocks from a reference video frame to blocks from a frame that the kernel is encoding. Each comparison generates a value indicating how similar the blocks are; a later scan of these values selects the best match. Thread blocks group together threads that inspect the same block of the current frame.

Using texture memory to hold the reference frame, the kernel takes advantage of locality and hardware support for boundary-value calculation that software would otherwise need to perform. However, the latency of texture memory is exposed to the kernel. The fastest kernel from H.264 that we've developed spends 20 percent of its time waiting for texture memory. Even so, the use of texture memory improves kernel performance by 2.8 times over storing the reference frame in global memory. Because texture memory's latency often impacts performance, it's usually the fallback when an algorithm's memory behavior strongly matches the hardware capabilities of the texture memory.

An SM's shared memory is useful for data that multiple threads in a thread block can share and reuse to eliminate redundant accesses to the global memory. Its contents only exist during thread-block execution and are discarded when the thread block completes. Kernels that read or write a known range of global memory with spatial or temporal locality can employ shared memory as a software-managed cache. Such caching potentially reduces global-memory bandwidth demands and improves performance.

Figure 5 shows an example of a tiled matrix-multiplication kernel. Unlike the original code shown earlier, threads in a 16 × 16 thread block cooperatively load two input tiles into shared memory. This amortizes the cost of global-memory access because each thread performs 1/16 the number of global loads. Threads calculate a partial dot product with the values in the tiles and repeat the process. Barrier synchronization ensures that the threads use and discard correct values when appropriate. Because the kernel performs tile loading separately from the subsequent computation, global-memory loads can be reorganized to achieve coalescing by having a half-warp's 16 threads load a row of 16 matrix elements.

Applications

As with other parallel computers, a parallel al-

```
Csub = 0;
for (...) {
  //Allocate arrays for tiles
  _shared_float AS[16][16];
  _shared_float BS[16][16];

  // Cooperatively load two import
  // tiles into shared memory
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16; * widthB;

  // Synchronous to ensure safety
  _syncthreads();

  // Calculate partial dot product
  for (i = 0; i < 16; 1++)
    {
      Csub += As[ty][i]
        * Bs[i][tx];
    }
  // Synchronize again before
  // loading new tiles
  _syncthreads();
}
C[c] = Csub;
```

Figure 5. Tiled matrix-multiplication kernel. It shared memory to reduce redundant global-memory accesses made by multiple threads to reduce global-memory bandwidth demands and improve performance.

gorithm's characteristics can significantly affect the GeForce 8800's performance. The algorithm should be decomposable into many independent threads, and to accommodate the GPU's SIMD-like execution behavior, the computational work should be nearly uniform across all threads. The algorithm should have a high ratio of computation to global-memory accesses, keeping itself busy with many arithmetic operations for each element of data that it consumes. Table 1 lists a group of applications that pass this first set of criteria, showing the breadth of scientific and engineering problems that can take advantage of parallel processing.

Our application selection is biased toward classes of problems such as linear algebra—Mat Mul and Saxpy—and stencil- and grid-based computations—the lattice Boltzmann method (LBM), MRI-Q, MRI-FHD, and the Coulombic Potential (CP)—that operate on data structures with very simple array layouts. Predictable, uniform array accesses conform much more easily to the necessary access patterns to utilize the GPU's hardware. Although the hardware permits random access reads and writes, kernels that extensively use random access to memory other than texture are unlikely to muster improvement over an optimized CPU version. Our selection also satisfies the requirement for SIMD-like execution. Kernel code ranges from 31 to 280 lines of code per application. Applications range from a few hundred to a few thousand lines of code (excluding comments and white space). H.264 is an outlier with roughly 35,000 lines of code.

We executed applications on a base system consisting of an Intel Core2 Extreme Quad running at 2.66 GHz with 4 Gbytes of main memory. For all applications except H.264, the unparallelized kernel code occupies the majority of the execution time; for H.264, it occupies 35 percent. Kernel speedup depends on CPU-only execution. We measure the CPU execution time of matrix multiplication using the Intel Math Kernel Library 8.0, which is multithreaded. CP, MRI-Q, and MRI-FHD were vectorized with SSE, and MRI-Q and MRI-FHD were additionally parallelized on four CPU cores. For the remaining applications, CPU times represent single-threaded C code. Speedup isn't directly comparable between kernels because of the different degrees of CPU parallelism.

To help explain kernel speedup, we state the ratio of global-memory access time to computation time to indicate approximately how memory intensive an application is. We compute the ratio from the lower bounds on memory-access time, assuming peak bandwidth, and on computation time, assuming no execution stalls. We also state the architectural bottlenecks that appear to limit the implementation from achieving a higher performance. The current generation of CUDA profiling tools can count undesirable events such as uncoalesced global loads, but individual contributions to a kernel's execution time can't be directly measured. We've found it useful to identify bottlenecks by comparing different implementations of the same kernel. If performance is insensitive to changes in the number of global-memory accesses, for example, then global access latency isn't a bottleneck. This general strategy provides the detailed insight into performances explained later.

Some performance differences are due to the GPU's hardware support for specific operations. In the MRI applications, for instance, a substantial number of executed operations are trigonometry functions. The GPU's hardware trigonometric operations are faster than even CPU fast math libraries, which accounts for approximately 30 percent of the speedup. Hardware trigonometry operations also speed up the Rys polynomial equation solver (RPES)

Table 1. Application suites.

Kernel	Description	Global-memory-to-computation-time ratio	Architectural bottlenecks	Kernel speedup on GPU
Mat Mul	Multiplication of two 4k × 4k dense matrices.	0.016	Instruction issue	9.3×
H.264	Modified version of the 464.h264ref benchmark from SPEC CPU2006 is an H.264 (MPEG-4 AVC) video encoder; the kernel computes SADs (sums of absolute differences) for full search motion estimation.	0.006	Register file capacity and cache latencies	12.23×
LBM	Modified version of the 470.lbm benchmark from SPEC CPU2006 that uses the lattice Boltzmann method for simulating 3D fluid dynamics; the program has been changed to use single-precision floating point (FP).	0.066	Shared-memory capacity	30.6×
RPES	Rys polynomial equation solver; calculates two-electron repulsion integrals, which are a subproblem of molecular dynamics.	0.01	Instruction issue	205×
PNS	Petri net simulation; distributed system's mathematical representation simulation.	0.241	Global-memory latency and branch divergence	26.8×
Saxpy	Single-precision FP implementation of Saxpy from high-performance Linpack; used as part of a Gaussian elimination routine.	0.375	Global-memory bandwidth	13.5×
MRI-Q	Computation of a matrix **Q**, representing the scanner configuration; used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	0.008	Instruction issue	31.2×
MRIFHD	Computation of an image-specific matrix FHD; used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	0.006	Instruction issue	11×
CP	Computation of electric potential in a volume containing point charges; based on direct Coulomb summation.[10]	0.0005	Instruction issue	64×

and CP. A different compute resource, the texture unit's built-in clamping and interpolation, speeds up H.264.

When a kernel attains sufficient throughput in the GPU's memory system and enough simultaneously executing threads to hide memory-access latency, the kernel's execution rate is limited only by the maximum rate at which an SM can issue instructions. This situation maximizes the time that the GPU spends performing useful computation and is the most desired outcome. Matrix multiply, MRI-Q, MRI-FHD, and CP are in this category. They satisfy most memory accesses using shared or constant memory in ways that don't cause stalls.

Every thread in CP, MRI-Q, and MRI-FHD reads the same sequence of memory addresses within the primary data-parallel loop. Placing the data in constant memory exploits the hardware's ability to service many accesses to the same address with a single-cache access, as described earlier in the section on memory spaces. These kernels are able to run without waiting for memory accesses.

Matrix multiplication exploits tiling to achieve high memory throughput. This kernel has regular access patterns to its input data and uses the data repeatedly. The input arrays are decomposed into tiles, which are explicitly copied into shared memory as needed. Once

copied, the data is accessible with single-cycle access time. The tile size must be a multiple of 16 to achieve global-memory coalescing. With a 16 × 16 block size from each input matrix, each value is reused from shared memory 16 times, which is enough to shift the bottleneck from global-memory bandwidth to instruction issue. We use a 16 × 16 tile from one matrix, employing register tiling[11] to distribute additional computation among threads to further reduce the instruction count.

The main computational kernel of RPES is the most FP intensive out of all our applications. Its main loop performs eight table lookups from texture memory followed by 37 to 105 FP operations (some are conditionally executed). Fifteen percent of its execution time is spent reading from texture memory. Although the GPU's memory system doesn't feed this kernel as efficiently as it does the previously mentioned kernels, this kernel is issue-limited due to the great quantity of FP computation.

RPES's irregular memory-access pattern is a consequence of how it was adapted to CUDA's thread-block and grid structure. RPES computes two-electron repulsion integrals, used to characterize the properties of atomic bonds from first principles. Electron distributions are represented as linear combinations of basis functions. A contribution to the electron repulsion integral is computed for every combination of four basis functions, giving the algorithm $O(n^4)$ complexity. On the CPU, this computation is performed in eight nested loops: the outer four loops traverse electron shells, and the inner four traverse basis functions. The number of basis functions varies for different electron shells; consequently, the inner loops have varying trip counts, and there's no simple way to map CUDA thread IDs to CPU loop iterations. To distribute computation to GPU threads, our implementation computes an assignment of work to thread blocks prior to launching the GPU kernel. An array holds the atoms' and electron shells' indices that each thread block should process. Within a thread block, each thread uses these indices and its own ID to fetch and process a single set of basis functions. The final distribution of computation has little in common with the distribution of data that it accesses, resulting in an irregular memory-access pattern. Texture memory accommodates these irregular accesses with significant but acceptable latency.

Capacity-limited kernels are based on algorithms that can mitigate the impact of memory-access latency or bandwidth through hardware- or software-managed caching but would require a larger shared memory or cache to eliminate it. H.264 uses register tiling to reduce the number of texture accesses by half. The size of the register file constrains the extent of register tiling, and the developer has to experimentally determine an acceptable tile size. At larger tile sizes, an SM is occupied by threads that are individually more efficient but fewer in number, trading a reduction in texture accesses for an increase in the amount of time per access that's not covered by useful work from other threads.

LBM is a time-stepped stencil simulation that's computation-bound on a CPU, but bandwidth-bound on a GPU due to shared-memory capacity limitations. The simulation uses a 3D lattice of cells, each storing 19 data elements representing fluid flow through the cell. We transformed the original array of structures to 19 arrays to meet the requirements of global-memory coalescing. The stencil for LBM is the pattern of 18 neighbors of any given cell (six sharing a face and 12 sharing an edge with the cell) that the algorithm uses to compute the cell's new fluid flow at each time step. The kernel reads the entire simulation lattice from global memory and writes back the updated lattice in each time step. This data transfer saturates the GPU's global-memory bandwidth.

A well-known multiprocessor technique for reducing communication in stencil computations is to compute multiple time steps of a sublattice using a private data area (GPU shared memory) before writing the results back to main memory (GPU global memory). However, each additional time step computed locally requires that a larger ghost area of the lattice be saved in shared memory. For this optimization to save bandwidth, the size of the ghost area must be small compared to the size of the sublattice. The largest sublattice that shared memory can hold for LBM is a cube of 4^3 cells with a cell envelope (144 ghost cells, excluding eight corner cells that aren't needed); the envelope is more than twice the size of the sublattice, nullifying any potential bandwidth savings.

Bandwidth-limited kernels such as Saxpy overwhelm the memory system's ability to supply data because they don't reuse data. Saxpy uses each input value only once, and performs only one FP multiply-accumulate operation for every two memory accesses. Adjacent threads access adjacent data values, enabling the kernel to exploit spatial locality.

Memory latency can also limit kernels. An example would be a kernel that relies heavily on pointer chasing or indirect array access to data

stored in global memory. Because such a kernel is unlikely to outperform a CPU, we avoided selecting applications with data-dependent memory-access patterns.

The main computational kernel of our Petri net simulation (PNS) algorithm is large and complex, making it harder to pinpoint a single bottleneck. A PNS is a Monte Carlo simulation of a Petri net. Each thread block runs an independent simulation with a different random number generator seed. The number of thread blocks is limited by the amount of simulation state that can be stored in global memory. The random number generator has a fixed degree of concurrency that determines the number of threads per block. Together, these two limits cap the amount of thread-level parallelism in the kernel. Limited thread parallelism exposes some of the latency of updating the simulation state in global memory. Besides suffering from limited concurrency, execution is also slowed down as threads within a block take different control-flow paths, some of which are data-dependent. Frequent explicit barrier synchronization between threads of a single block along with high memory-to-computation-cycle ratios can also degrade PNS kernel performance.

Application Optimization

In general, we obtained significant kernel and application speedup across our suite (see Table 1). Compute-intensive kernels with relatively few global-memory accesses achieve very high performance. Even kernels that don't have high compute-to-global-memory access ratios still achieve respectable performance increases because of the GeForce 8800's ability to run a large number of threads simultaneously. However, the performance shown in the table was generally obtained after a significant amount of hand tuning. We've already mentioned some of the memory optimizations that improve performance—we discuss several others here.

The distribution of computation threads is generally decided in the initial phases of program optimization, but it has long-reaching effects. Thread granularities might be too small to take advantage of data reuse or too large to fit many threads onto the hardware simultaneously. Loop interchange—changing how loops nest—is a useful strategy prior to distributing work because it can alter the threads' data-access patterns and enable better memory usage; MRI kernels are a prime example of this. Although in the optimized CPU code, the outer loop traverses sample values, in the GPU kernel, the loops are interchanged so that the inner loop traverses sample values, allowing them to be placed in constant memory.

Loop unrolling and other classic compiler optimizations such as those found in Kennedy and Allen's work[12] can have unexpected results, but, in general, local optimizations on the most frequently executed parts of the code have beneficial effects. These optimizations directly target the instruction-issue bottleneck by reducing the number of executed operations or through strength reduction. In H.264 and matrix multiplication, complete unrolling of the innermost loop obtains significant performance increase, as does register tiling.[11]

When attempted optimizations have negative effects, the most common cause is that they increase the number of registers per thread as a side effect, forcing the GeForce 8800 to schedule fewer thread blocks per SM and thus degrading performance. The cases where this is most often seen are common-subexpression elimination and redundant-load elimination. Even relatively simple instruction scheduling by CUDA's runtime can change the live ranges of variables and increase register usage. Researchers have investigated register-pressure-sensitive code-scheduling algorithms and optimization strategies in the context of instruction-level parallelism-extracting compilers; additional research is necessary to apply these strategies to massively threaded environments such as CUDA.

GPUs have been available for more than a decade but only recently have they had a programming model and architecture that opens them up as more general computing platforms. We've shown that a variety of kernels achieve significant speedups on a system equipped with a GPU. However, the optimization process requires strong knowledge of both application and architecture and still requires major time and effort to achieve near-peak performance. We and other researchers are working on better techniques, tools, and compilers to make mapping applications to these systems easier in the future. CiSE

References

1. J. Owens, *GPU Gems 2*, Addison-Wesley, 2005, pp. 457–470.
2. M.J. Atallah, ed., *Algorithms and Theory of Computation Handbook*, CRC Press, 1998.
3. S. Ryoo et al., "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, ACM Press, 2008, pp. 73–82.

4. OpenMP Architecture Rev. Board, "OpenMP Application Program Interface," May 2005; www.openmp.org/mp-documents/spec25.pdf.
5. I. Buck et al., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM SIGGRAPH 2004 Papers*, ACM Press, 2004, pp. 777–786.
6. D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General Purpose Uses," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2006, pp. 325–335.
7. L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics*, vol. 27, Aug. 2008, pp. 1–15.
8. AMD, *R600-Family Instruction Set Architecture*, tech. rep., Advanced Micro Devices, May 2007.
9. S.S. Stone et al. "Accelerating Advanced MRI Reconstruction using GPUs," *ACM Computing Frontiers Conf. 2008*, ACM Press, 2008, pp. 251–260.
10. J.E. Stone et al., "Accelerating Molecular Modeling Applications with Graphics Processors," *J. Computational Chemistry*, vol. 28, Dec. 2007, pp. 2618–2640.
11. D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *ACM SIGPLAN Notices*, vol. 9, no. 4, 2004, pp. 328–342.
12. K. Kennedy and J.R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, Morgan Kaufmann, 2002.

Wen-Mei Hwu *is a professor and holds the Sanders-AMD Endowed Chair of the Electrical and Computer Engineering Department at the University of Illinois. His research interests include compiler technology and programming techniques for parallel systems. Hwu has a PhD in computer science from the University of California, Berkeley. Contact him at w-hwu@uiuc.edu.*

Christopher Rodrigues *is a graduate research assistant at the University of Illinois. His research interests include language and compiler support for parallelization and memory safety. Rodrigues has an MS in electrical and computer engineering from the University of Illinois. Contact him at cirodrig@illinois.edu.*

Shane Ryoo *is a senior software engineer at ZeroSoft. His research interests include application-specific compilation and holistic optimization. Ryoo has a PhD in electrical and computer engineering from the University of Illinois. Contact him at sryoo@illinoisalumni.org.*

John Stratton *is a graduate research assistant at the University of Illinois. His research interests include parallel programming models, parallel application portability, and programming productivity. Stratton has a BS in computer engineering from the University of Illinois. Contact him at stratton@illinois.edu.*