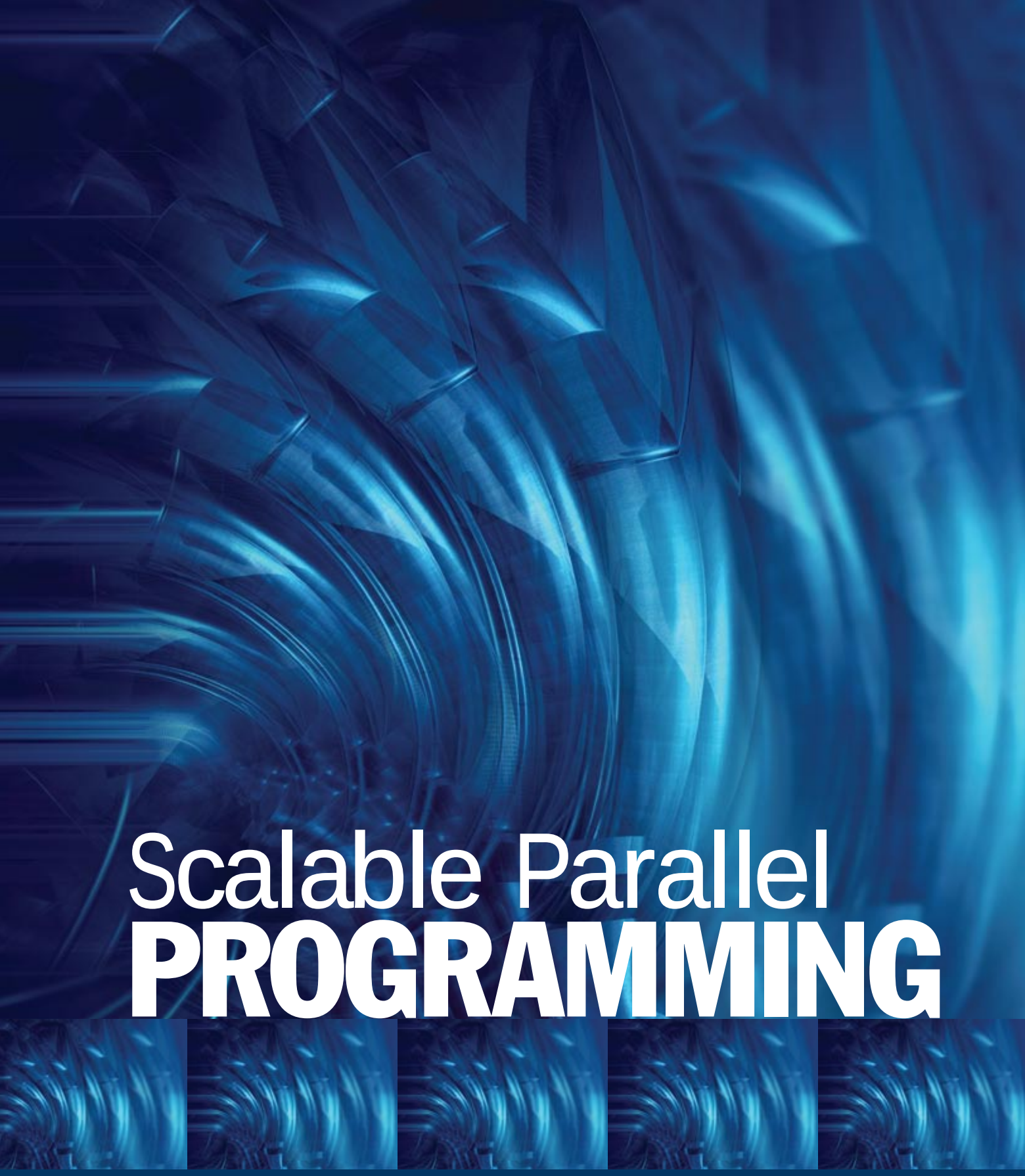
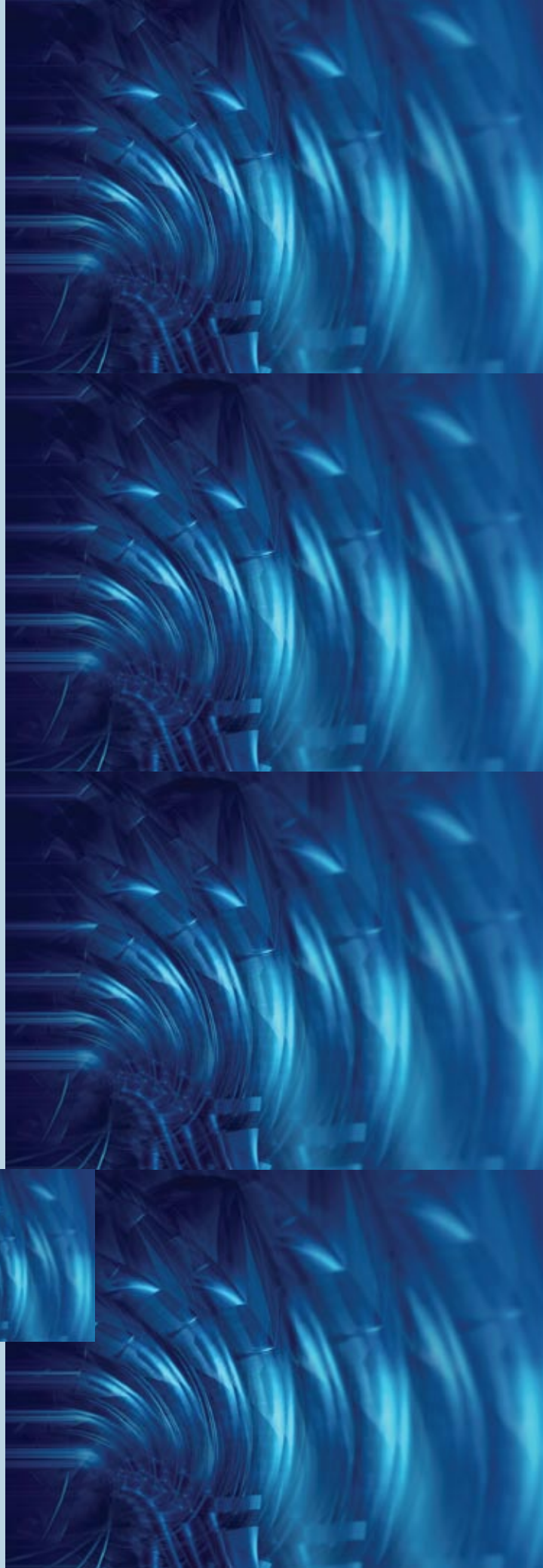


Scalable Parallel PROGRAMMING

JOHN NICKOLLS, IAN BUCK, AND
MICHAEL GARLAND, NVIDIA,
KEVIN SKADRON, UNIVERSITY OF VIRGINIA



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 application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

According to conventional wisdom, parallel programming is difficult. Early experience with the CUDA[1,2] scalable parallel programming model and C language, however, shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the new Tesla unified graphics and computing architecture (described in the GPU sidebar) run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA

with CUDA

Is CUDA the parallel programming model that application developers have been waiting for?

Scalable Parallel PROGRAMMING with CUDA

model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs.[3]

CUDA provides three key abstractions—a hierarchy of thread groups, shared memories, and barrier synchronization—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into

UNIFIED GRAPHICS AND COMPUTING GPUS

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU (graphics processing unit) has evolved into a highly parallel, multithreaded, manycore processor. It is designed to efficiently support the graphics shader programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels at fine-grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently.

The tremendous raw performance of modern GPUs has led researchers to explore mapping more general non-graphics computations onto them. These GPGPU (general-purpose computation on GPUs) systems have produced some impressive results, but the limitations and difficulties of doing this via graphics APIs are legend. This desire to use the GPU as a more general parallel computing device motivated NVIDIA to develop a new unified graphics and computing GPU architecture and the CUDA programming model.

GPU COMPUTING ARCHITECTURE

Introduced by NVIDIA in November 2006, the Tesla unified graphics and computing architecture[1,2] significantly extends the GPU beyond graphics—its massively multithreaded processor array becomes a highly efficient unified platform for *both* graphics and general-purpose parallel computing applications. By scaling the number of processors and memory partitions, the Tesla architecture spans a wide market range—from the high-performance enthusiast GeForce 8800 GPU and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs. Its computing features enable straightforward programming of the GPU cores in C with CUDA. Wide availability in laptops, desktops, workstations, and servers, coupled with C programmability and CUDA software, make the Tesla architecture the first ubiquitous supercomputing platform.

The Tesla architecture is built around a scalable array of multithreaded SMs (streaming multiprocessors). Current GPU implementations range from 768 to 12,288 concurrently executing threads. Transparent scaling across this wide range of available parallelism is a key design goal of both the GPU architecture and the CUDA programming model. Figure A shows a GPU with 14 SMs—a total of 112 SP (streaming processor) cores—interconnected with four external DRAM partitions. When a CUDA program on the host CPU invokes a kernel grid, the CWD (compute work distribution) unit enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

An SM consists of eight scalar SP cores, two SFUs (special function units) for transcendentals, an MT IU (multithreaded instruction unit), and on-chip shared memory. The SM creates, manages, and executes up to 768 concurrent threads in hardware with zero scheduling overhead. It can execute as many as eight CUDA thread blocks concurrently, limited by thread and memory resources. The SM implements the CUDA __syncthreads() barrier synchronization intrinsic with a single instruction. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing a new thread to be created to compute each vertex, pixel, and data point.

To manage hundreds of threads running several different programs, the Tesla SM employs a new architecture we call

finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

THE CUDA PARADIGM

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel *kernel*s, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A *grid* is a set of thread blocks that may each be executed independently and thus may execute in parallel.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks

SIMT (single-instruction, multiple-thread).[3] The SM maps each thread to one SP scalar core, and each scalar thread executes independently with its own instruction address and register state. The SM SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. (This term originates from weaving, the first parallel thread technology.) Individual threads composing a

Continued on the next page

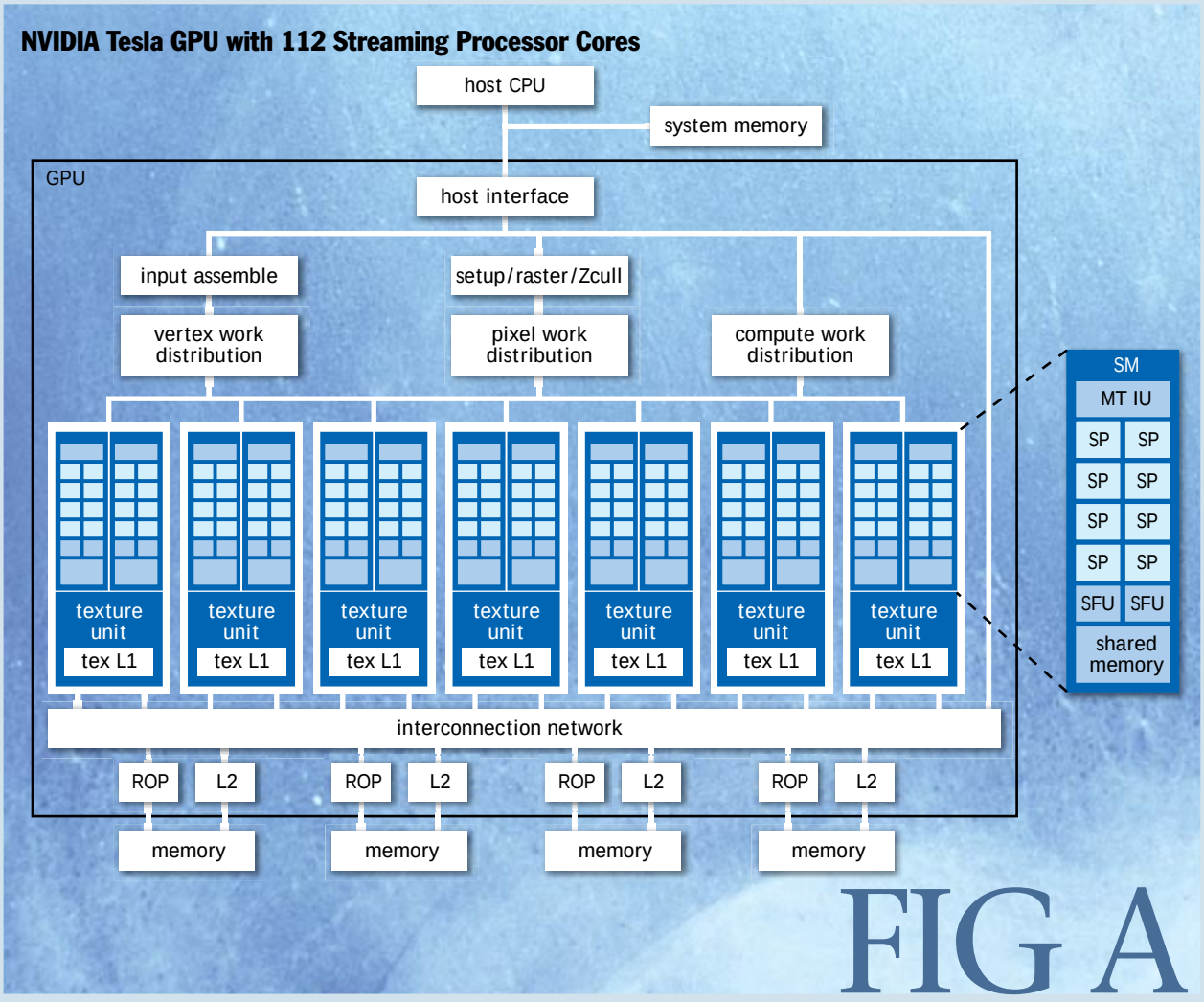


FIG A

Scalable Parallel **PROGRAMMING** with **CUDA**

making up the grid. Each thread is given a unique *thread ID* number threadIdx within its thread block, numbered 0, 1, 2, …, blockDim–1, and each thread block is given a unique *block ID* number blockIdx within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.

As a very simple example of parallel programming, suppose that we are given two vectors x and y of n floating-point numbers each and that we wish to compute the result of $y \leftarrow ax + y$, for some scalar value a. This is the

SIMT warp start together at the same program address but are otherwise free to branch and execute independently. Each SM manages a pool of 24 warps of 32 threads per warp, a total of 768 threads.

Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. As a result, the Tesla-architecture GPUs are dramatically more efficient and flexible on branching code than previous-generation GPUs, as their 32-thread warps are much narrower than the SIMD (single-instruction multiple-data) width of prior GPUs.

SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

A thread's variables typically reside in live registers. The 16KB SM shared memory has very low access latency and high bandwidth similar to an L1 cache; it holds CUDA per-block __shared__ variables for the active thread blocks. The SM provides load/store instructions to access CUDA __device__ variables in GPU external DRAM. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. Because global memory latency can be hundreds of processor clocks, CUDA programs copy data to shared memory when it must be accessed multiple times by a thread block. Tesla load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimizations. The large thread count in each SM, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM. The latest Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

CUDA applications perform well on Tesla-architecture GPUs because CUDA's parallelism, synchronization, shared memories, and hierarchy of thread groups map efficiently to features of the GPU architecture, and because CUDA expresses application parallelism well.

REFERENCES

1. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28(2).
2. Nickolls, J. 2007. NVIDIA GPU parallel computing architecture. In *IEEE Hot Chips* 19 (August 20), Stanford, CA; http://www.hotchips.org/archives/hc19/.
3. See reference 1.

so-called saxpy kernel defined by the BLAS (basic linear algebra subprograms) library. The code for performing this computation on both a serial processor and in parallel using CUDA is shown in figure 1.

The __global__ declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function-call syntax

```
kernel<<<dimGrid, dimBlock>>>(... parameter list ...);
```

where dimGrid and dimBlock are three-element vectors of type dim3 that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to 1.

In the example, we launch a grid that assigns one thread to each element of the vectors and puts 256 threads in each block. Each thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. The serial and parallel versions of this code are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

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);
```

FIG 1

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling, and termination are handled for the programmer by the underlying system. Indeed, a Tesla-architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a barrier by calling the __syncthreads() intrinsic. This guarantees that no thread participating in the barrier can proceed until all participating threads have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by participating threads before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Since threads in a block may share local memory and synchronize via barriers, they will reside on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. This virtualizes the processing elements and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. This also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example.

This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary

Scalable Parallel PROGRAMMING with CUDA

number of cores, as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock.

An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently given sufficient hardware resources. Dependent grids execute sequentially, with an implicit inter-kernel barrier between them, thus guaranteeing that all blocks of the first grid will complete before any block of the second dependent grid is launched.

Threads may access data from multiple memory spaces during their execution. Each thread has a private *local* memory. CUDA uses this memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a *shared* memory visible to all threads of the block that has the same lifetime as the block. Finally, all threads have access to the same *global* memory. Programs declare variables in shared and global memory with the __shared__ and __device__ type qualifiers. On a Tesla-architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can, therefore, provide for high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure 2 diagrams the nested levels of threads, thread blocks, and grids of thread blocks. It shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

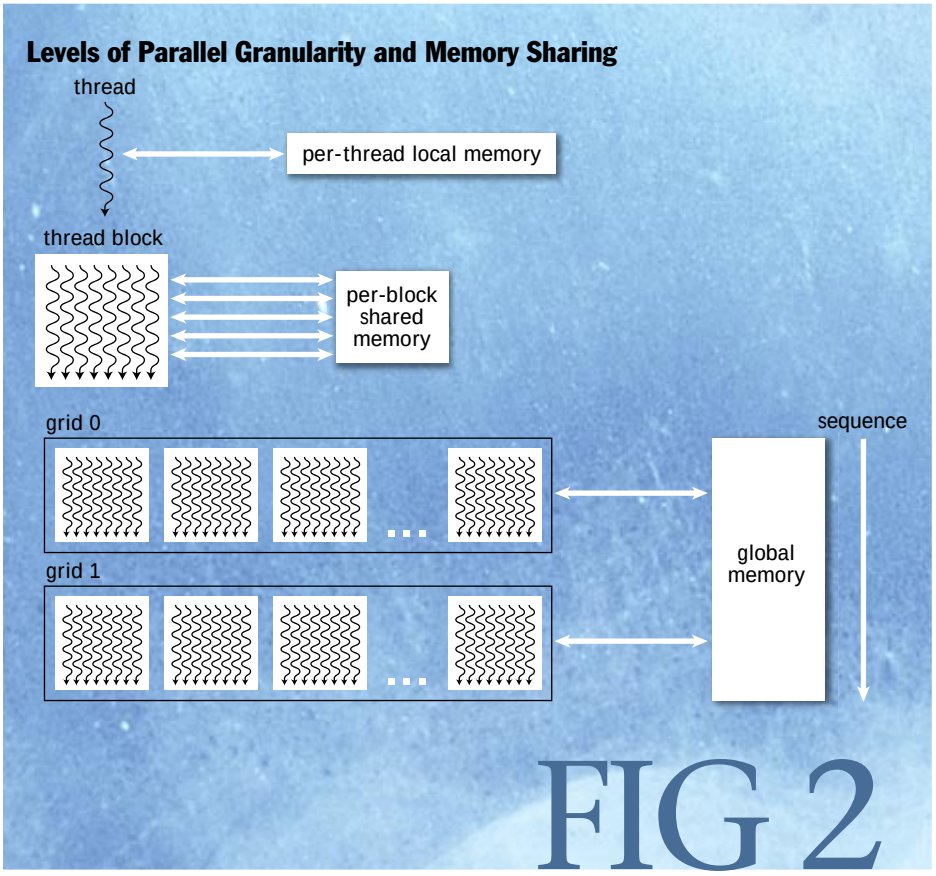


FIG 2

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as cudaMalloc() and cudaFree(). Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use cudaMemcpy() to copy data between the allocated space and the host system memory.

The CUDA programming model is similar in style to the familiar SPMD (single-program multiple-data) model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. CUDA, however, is more flexible than most real-

izations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads.

Figure 3 shows an example of a SPMD-like CUDA code sequence. It first instantiates kernelF on a 2D grid of 3×2 blocks where each 2D thread block consists of 5×3 threads. It then instantiates kernelG on a 1D grid of four 1D thread blocks with six threads each. Because kernelG depends on the results of kernelF, they are separated by an inter-kernel synchronization barrier.

The concurrent threads of a thread block express fine-grained *data* and *thread* parallelism. The independent thread blocks of a grid express coarse-grained *data* parallelism. Independent grids express coarse-grained *task* parallelism. A *kernel* is simply C code for one thread of the hierarchy.

RESTRICTIONS

When developing CUDA programs, it is important to understand the ways in which the CUDA model is restricted, largely for reasons of efficiency. Threads and thread blocks may be created only by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla architecture implements *hardware* management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread-block level, but blockwide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between thread blocks within the same kernel grid is not allowed—they must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any

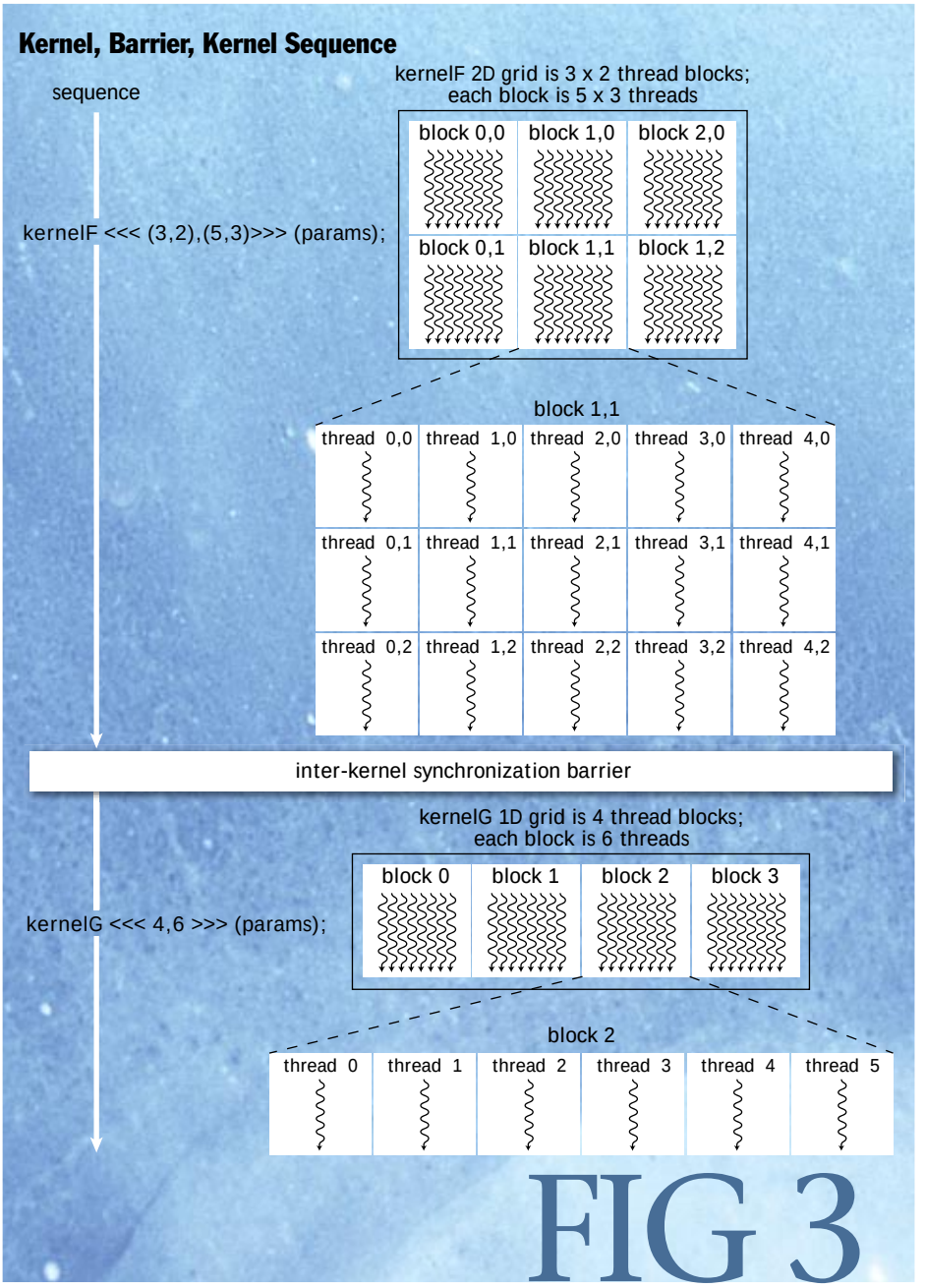


FIG 3

Scalable Parallel PROGRAMMING with CUDA

order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks. However, multiple thread blocks can coordinate their work using atomic operations on global memory (e.g., to manage a data structure).

Recursive function calls are not allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects. Of course, problems large enough to need a GPU performance boost amortize the overhead better than small problems.

RELATED WORK

Although the first CUDA implementation targets NVIDIA GPUs, the CUDA abstractions are general and useful for programming multicore CPUs and scalable parallel systems. Coarse-grained thread blocks map naturally to separate processor cores, while fine-grained threads map to multiple-thread contexts, vector operations, and pipelined loops in each core. Stratton et al. have developed a prototype source-to-source translation framework that compiles CUDA programs for multicore CPUs by mapping a thread block to loops within a single CPU thread. They found that CUDA kernels compiled in this way perform and scale well.[4]

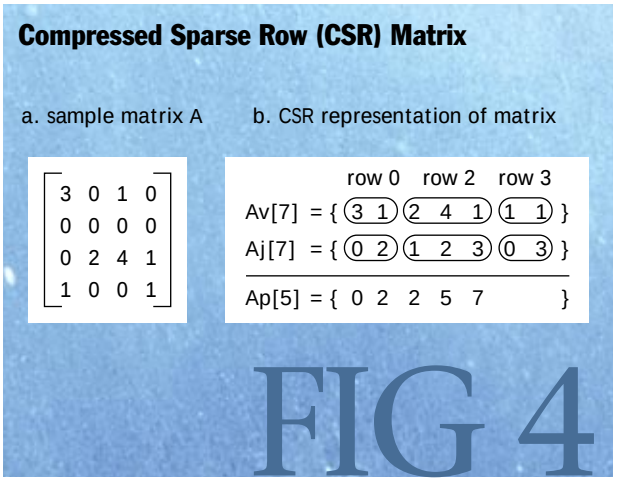
CUDA uses parallel kernels similar to recent GPGPU programming models, but differs by providing flexible thread creation, thread blocks, shared memory, global memory, and explicit synchronization. Streaming languages apply parallel kernels to data records from a stream. Applying a stream kernel to one record is analogous to executing a single CUDA kernel thread, but stream programs do not allow dependencies among kernel threads, and kernels communicate only via FIFO (first-in, first-out) streams. Brook for GPUs differentiates between FIFO input/output streams and random-access *gather* streams, and it supports parallel reductions. Brook is a good fit for earlier-generation GPUs with random access texture units and raster pixel operation units.[5]

Pthreads and Java provide fork-join parallelism but are not particularly convenient for data-parallel applications. OpenMP targets shared memory architectures with parallel execution constructs, including "parallel for" and teams of coarse-grained threads. Intel's C++ Threading Building Blocks provide similar features for multicore CPUs. MPI targets distributed memory systems and uses message passing rather than shared memory.

CUDA APPLICATION EXPERIENCE

The CUDA programming model extends the C language with a small number of additional parallel abstractions. Programmers who are comfortable developing in C can quickly begin writing CUDA programs.

In the relatively short period since the introduction of CUDA, a number of real-world parallel application codes have been developed using the CUDA model. These include FHD-spiral MRI reconstruction,[6] molecular dynamics,[7] and n-body astrophysics simulation.[8] Running on Tesla-architecture GPUs, these applications were able to achieve substantial speedups over alternative implementations running on serial CPUs: the MRI reconstruction was 263 times faster; the molecular dynamics code was 10–100 times faster; and the n-body simulation was 50–250 times faster. These large speedups are a result of the highly parallel nature of the Tesla architecture and its high memory bandwidth.



EXAMPLE: SPARSE MATRIX-VECTOR PRODUCT

A 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. SpMV (sparse matrix-vector multiplication) 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 here, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient[9] 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 savings in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the CSR (compressed sparse row) 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

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // column indices for row
                   float *Av,        // non-zero 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;
}
```

FIG 5

```
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);
   }
}
```

FIG 6

```
__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);
   }
}
```

FIG 7

Scalable Parallel PROGRAMMING with CUDA

elements records the extent of each row in the previous arrays; the entries for row *i* in Aj and Av extend from index Ap[i] up to, but not including, index Ap[i+1]. This implies that Ap[0] will always be 0 and Ap[n] will always be the number of nonzero elements in the matrix. Figure 4 shows an example of the CSR representation of a simple matrix.

Given a matrix *A* in CSR form, we can compute a single row of the product $y = Ax$ using the multiply_row() procedure shown in figure 5.

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 shown in figure 6.

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*. Figure 7 shows the code for this kernel. 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 is computed from the block and thread indices assigned to each thread. Second, we have a conditional that evaluates a row product only 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).

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like the code in figure 8.

The pattern that we see here is a 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.

```
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);
```

FIG 8

```
__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;
  }
}
```

FIG 9

CACHING IN SHARED MEMORY

The SpMV algorithms outlined here are fairly simplistic. We can make a number of optimizations in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking.[10] The parallel kernels can also be reimplemented in terms of data-parallel *scan* operations.[11]

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, shown in figure 9.

In the context of sparse matrix multiplication, we observe that several rows of *A* may use a particular array element x[i]. In many common cases, and particularly when the matrix has been reordered, the rows using x[i] will be rows near row *i*. We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows *i* through *j* will load x[i] through x[j] into its shared memory. We will unroll the multiply_row() loop and fetch elements of x from the cache whenever possible. The resulting code is shown in figure 9. 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 in figure 9, even this fairly simple use of shared memory returns a roughly 20 percent 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.

EXAMPLE: PARALLEL REDUCTION

Suppose that we are given a sequence of *N* integers that must be combined in some fashion (e.g., a sum). This occurs in a variety of algorithms, linear algebra being a common example. On a serial processor, we would write a simple loop with a single accumulator variable to construct the sum of all elements in sequence. On a parallel machine, using a single accumulator variable would create a global serialization point and lead to very poor performance. A well-known solution to this problem is the so-called *parallel reduction* algorithm. Each parallel thread sums a fixed-length subsequence of the input. We then collect these partial sums together, by summing

```
__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. See attached figure.
   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]);
}
```

FIG 10

Scalable Parallel **PROGRAMMING** with **CUDA**

pairs of partial sums in parallel. Each step of this pair-wise summation cuts the number of partial sums in half and ultimately produces the final sum after $\log_2 N$ steps. Note that this implicitly builds a tree structure over the initial partial sums.

In the example shown in figure 10, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length one). 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. We can achieve this in parallel by summing values in a tree-like pattern. The loop in this kernel implicitly builds a summation tree over the input elements. The action of this loop for the simple case of a block of eight threads is illustrated in figure 11. The steps of the loop are shown as successive levels of the diagram and edges indicate from where partial sums are being read.

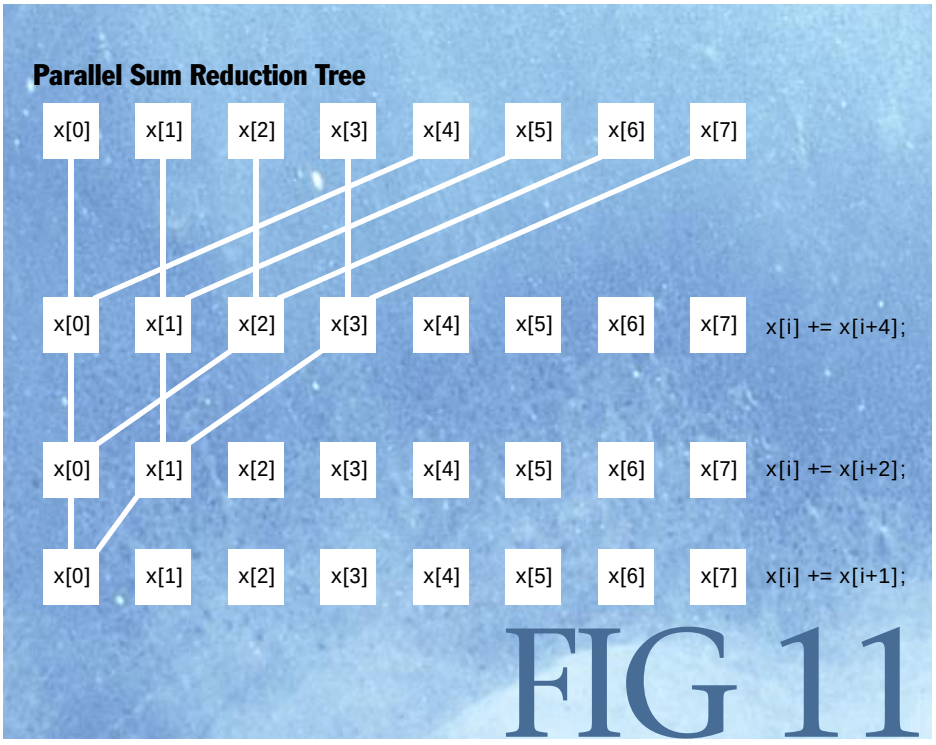


FIG 11

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 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 architecture is to use `atomicAdd()`, 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.

THE DEMOCRATIZATION OF PARALLEL PROGRAMMING

CUDA is a model for parallel programming that provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications. In fact, CUDA is an excellent programming environment for teaching parallel programming. The University of Virginia has used it as just a short, three-week module in an undergraduate computer architecture course, and students were able to write a correct k-means clustering program after just three lectures. The University of Illinois has successfully taught a semester-long parallel programming course using CUDA to a mix of computer science and non-computer science majors, with students obtaining impressive speedups on a variety of real applications, including the previously mentioned MRI reconstruction example.

CUDA is supported on NVIDIA GPUs with the Tesla unified graphics and computing architecture of the GeForce 8-series, recent Quadro, Tesla, and future GPUs.

The programming paradigm provided by CUDA has allowed developers to harness the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of 100 times or more on a variety of sophisticated applications.

The CUDA abstractions, however, are general and provide an excellent programming environment for multicore CPU chips. A prototype source-to-source translation framework developed at the University of Illinois compiles CUDA programs for multicore CPUs by mapping a parallel thread block to loops within a single physical thread. CUDA kernels compiled in this way exhibit excellent performance and scalability.[12]

Although CUDA was released less than a year ago, it is already the target of massive development activity—there are tens of thousands of CUDA developers. The combination of massive speedups, an intuitive programming environment, and affordable, ubiquitous hardware is rare in today's market. In short, CUDA represents a democratization of parallel programming. Q

REFERENCES

1. NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA.
2. NVIDIA. 2007. CUDA Programming Guide 1.1; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
3. Stratton, J.A., Stone, S. S., Hwu, W. W. 2008. M-CUDA: An efficient implementation of CUDA kernels on multicores. IMPACT Technical Report 08-01, University of Illinois at Urbana-Champaign, (February).
4. See reference 3.
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P. Brook for GPUs: Stream computing on graphics hardware. 2004. *Proceedings of SIGGRAPH* (August): 777-786; http://doi.acm.org/10.1145/1186562.1015800.
6. Stone, S.S., Yi, H., Hwu, W.W., Haldar, J.P., Sutton, B.P., Liang, Z.-P. 2007. How GPUs can improve the quality of magnetic resonance imaging. The First Workshop on General-Purpose Processing on Graphics Processing Units (October).
7. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K. 2007. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* 28(16): 2618–2640; http://dx.doi.org/10.1002/jcc.20829.
8. Nyland, L., Harris, M., Prins, J. 2007. Fast n-body simulation with CUDA. In *GPU Gems 3*. H. Nguyen, ed. Addison-Wesley.
9. Golub, G.H., and Van Loan, C.F. 1996. *Matrix Computations*, 3rd edition. Johns Hopkins University Press.
10. Buatois, L., Caumon, G., Lévy, B. 2007. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *Proceedings of the High-Performance Computation Conference (HPCC)*, Springer LNCS.
11. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D. 2007. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware* (August): 97–106.
12. See Reference 3.

Links to the latest version of the CUDA development tools, documentation, code samples, and user discussion forums can be found at: http://www.nvidia.com/CUDA.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JOHN NICKOLLS is director of architecture at NVIDIA for GPU computing. He was previously with Broadcom, Silicon Spice, Sun Microsystems, and was a cofounder of MasPar Computer. His interests include parallel processing systems, languages, and architectures. He has a B.S. in electrical engineering and computer science from the University of Illinois, and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

IAN BUCK works for NVIDIA as the GPU-Compute software manager. He completed his Ph.D. at the Stanford Graphics Lab in 2004. His thesis was titled "Stream Computing on Graphics Hardware," researching programming models and computing strategies for using graphics hardware as a general-purpose computing platform. His work included developing the Brook software tool chain for abstracting the GPU as a general-purpose streaming coprocessor.

MICHAEL GARLAND is a research scientist with NVIDIA Research. Prior to joining NVIDIA, he was an assistant professor in the department of computer science at the University of Illinois at Urbana-Champaign. He received Ph.D. and B.S. degrees from Carnegie Mellon University. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models.

KEVIN SKADRON is an associate professor in the department of computer science at the University of Virginia and is currently on sabbatical with NVIDIA Research. He received his Ph.D. from Princeton University and B.S. from Rice University. His research interests include power- and temperature-aware design, and manycore architecture and programming models. He is a senior member of the ACM.