Accelerate!

GPU Architecture and Applications
How to determine if an application is well-suited to GPU acceleration

Programming GPUs today
A survey of current low-level GPU programming models

Optimizing GPU Kernels
An in-depth look at optimizing matrix multiply on a GPU

A GPU and Accelerator Programming Model
Portable and incremental C and Fortran programming on x64+GPU platforms

Michael Wolfe

Compilers and More:
GPU Architecture and Applications

Accelerated computing is one of the most refreshing developments in high performance computing (HPC) in the last several years. Just when we appeared destined to count x86 cores like sheep, snoring through the latest MPI ping pong latency scores, along comes a paradigm promising orders of magnitude leaps in compute node performance. Will the promise be realized, or will accelerators succumb to the tick-tock march of the x86 penguins that has subsumed everything in its path except IBM's venerable POWER architecture? Is this a new fast track to desktop Petaflops, or just another HPC pretender?

Dr. Michael Wolfe is a technical Fellow and compiler engineer at The Portland Group with over 30 years of experience in research and development of optimizing compilers for HPC systems. In this series of articles originally published on HPCwire, Michael objectively explores the potential, power and perils of high performance computing on CPU+Accelerator platforms. Accelerated compute nodes can become the next HPC building block only if early successes on x64+GPU systems are generalized into a node architecture and programming model that is easily accessible to mainstream HPC application developers and users. Read on to learn whether this is possible, and how accelerated computing might have an impact on you, your applications and your organization in the years ahead.

Douglas Miles
Director
The Portland Group

One of the most exciting developments in parallel programming over the past few years has been the availability and advancement of programmable graphics cards. High end graphics cards cost less than a high end CPU and provide tantalizing peak performance approaching or exceeding one Teraflop. Since microprocessor peak performance tops out at about 25 Gigaflops/core (single precision), this potential, at such low cost, is worth exploring. Harnessing this performance, however, is problematical.

It's important to note that the GPUs powering the graphics cards are designed to do specific jobs very well. They are not designed as general purpose processors, and in fact will do a very poor job on many programs, even highly parallel applications. The key is to determine whether your application can fit into a programming model that maps well onto the GPU. I'm going to discuss the GPU architecture, but I'm going to start with an analogy, and probably stretch the analogy to the breaking point; let's discuss airline travel.

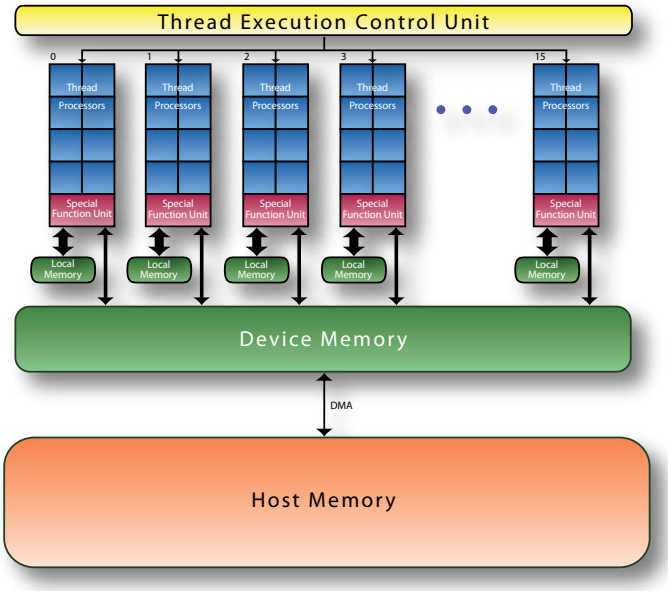
Suppose your job is to transport several dozen large tour groups between London and Seattle, each group with 30-60 members. Your most likely choice is to use jet aircraft, a flight of about 5,000 miles or 7,700 km. Going for the most parallelism, you could use a new Airbus A380 to move 600 people in about 9 hours. One problem you have with these jumbo jets is they don't fit at the main terminal, so you have to take an airport train out to the remote terminal where the plane is parked; the train can only carry so many people at a time, but let's be optimistic and say it will take 90 minutes to move everyone out to the remote gate and load them, and another 90 minutes at the other end for unloading. This gives us a 24 hour round trip. If you regularly fill the plane, this could be a good investment (though, at $300M, a bit more than a GPU). However, if you only have a half-load, it doesn't get them there twice as fast or at half the cost (though it does reduce the load/unload time). Measuring performance as passenger-miles, the performance comes from parallelism (many passengers), not from latency reduction.

Alternatively, you could opt for a few smaller planes, say two to four Boeing 787s. Each can carry about 200-250 passengers at the same speed, so you can move the same number of people at the same rate. You also have the advantage of parking at the main terminal, so you can reduce the passenger load/unload time to about half an hour, and your total round trip time is only 20 hours. Of course, you have to arrange for more crew, more landing slots at the airports, and so forth. Your total capital investment is about the same, but it gives you some flexibility. If you only have 200 people to move, you can leave all but one of the planes behind, saving on fuel and crew costs.

Or, perhaps you could invest in the (future) hypersonic transport, which some believe may be able to travel at speeds of Mach 6, 7-8 times faster than the current subsonic airliners. Assuming it takes time to get up to speed (and to slow down for landing), the total flight time might drop from 9 hours to 2.5 hours, or 7 hours round trip. If the capacity of your hypothetical hypersonic transport is 200 passengers, you can transport 600 people in each direction in just 21 hours. Even better, if you only have to transport 400 people, you can get the same work done in 2/3 the time. Of course, you are buying a higher cost transport, probably paying more for fuel, and so on.

Very high performance CPUs are more like the hypersonic transport; they are designed to give very high performance for small tasks, and give performance for large tasks using that speed. Multicore processors are more like the middle option, several smaller, lower-capacity devices, each quite capable, and you can save power by shutting one or more of them down. The GPU is more like the super-jumbo jet; it only gets high performance (passenger-miles) when you have lots of passengers. It doesn't do so well getting just one or a few passengers across the ocean.

So, now to GPU architecture. GPUs were originally hardwired for specific tasks; as transistor budgets and demand for flexibility grew, the hardware became more programmable. They still contain special hardware and functional units specific to graphics tasks, but I'll ignore those and



view today's GPUs as compute accelerators. A typical design, shown in the figure, is abstracted from the information in the NVIDIA documentation; I'll use both NVIDIA terms and more standard computer architecture descriptions of the various parts. The key to the performance is all those thread processors; in the figure, there are 8 thread processors in each of 16 multiprocessors, for 128 TPs total. NVIDIA delivers GPUs with up to 30 multiprocessors and 240 TPs. In each clock, each TP can produce a result, giving this design a very high peak performance rating.

Each multiprocessor executes in SIMD mode, meaning each thread processor in that multiprocessor executes the same instruction simultaneously. If one thread processor is executing a floating point add, they all are; if one is doing a fetch, they all are. Moreover, each instruction is quad-clocked, meaning the SIMD width is 32, even though there are only 8 thread processors. Unlike classical SIMD machines, there isn't a distinction between the scalar and parallel operations, or mono and poly operations, to borrow terms from C* and Dataparallel C, and Cn for the Clearspeed card. Instead, the model is that many scalar threads just happen to be executing in SIMD mode, something

NVIDIA calls SIMT execution. Careful orchestration of the 32 threads that execute in SIMD mode is necessary for best performance.

Stretching our analogy, think of each thread processor like a seat in our superjumbo jet, and each multiprocessor as a tour group. Imagine that when the flight attendant serves a meal, the whole tour group must be served at once (synchronously). The whole tour group must watch the same movie on the seat-back screens at the same time, or they all must read books at the same time, though some may be napping at any time. When one person wants to use the restroom, they all must go at once, even if not everyone needs to (note that this is a long latency operation, which will correspond to the long latency memory).

The multiprocessors themselves execute asynchronously, and without communication. This last point is quite important. In a multicore or multiprocessor system, the cores or processors can communicate through the memory. If one thread stores a value in variable A then sets a FLAG, the hardware will guarantee that another thread on the same or another core or processor will not see the updated FLAG without seeing the updated value for A. The hardware supports a memory model that preserves the store order. No such memory model is supported on the GPU; a program could store a set of values on one multiprocessor and read the same locations on another, but there is no guarantee that the value fetched will be consistent (in the formal sense) with the values stored. Relaxing the memory model allows the hardware to reorder the stores from a multiprocessor, allowing more throughput.

In this GPU, each multiprocessor has a special function unit, which handles infrequent, expensive operations, like divide, square root, and so on; it operates more slowly than other operations, but since it's infrequently used, it doesn't affect performance. There is a high bandwidth, low latency local memory attached to each multiprocessor. The threads executing on that multiprocessor can communicate among themselves using this local memory. In the current NVIDIA generation, the local memory is quite small (16KB).

There is also a large global device memory, up to 4GB in some models. This is physical, not virtual; paging is not supported, so all the data has to fit in the memory. The device memory has very high bandwidth, but high latency to the multiprocessors. The device memory is not directly accessible from the host, nor is the host memory directly visible to the GPU. Data from the host that needs to be processed by the GPU must be moved via DMA across an IO bus from the host to the device memory, and the results moved back (like loading jumbo jets using thin airport trains).

So, there is a hierarchy of parallelism on this GPU; threads executing within a multiprocessor can share and communicate using the local memory, while threads executing on different multiprocessors cannot communicate or synchronize. This hierarchy is explicit in the programming model as well. Parallelism comes in two flavors; outer, asynchronous parallelism between thread groups or thread blocks, and inner, synchronous parallelism within a thread block. All the threads of a thread block will always be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors.

Because of the high latency to the device memory, the multiprocessors are highly multithreaded as well. When one set of SIMD threads executes a memory operation, rather than stall, the multiprocessor will switch to execute another set of SIMD threads. The other SIMD threads may be part of the same thread block, or may come from a different thread block assigned to the same multiprocessor. Think of this as the flight attendant serving another tour group while one group gets up to visit the restrooms.

A GPU is designed as a throughput engine; it's designed to get a lot of work done on a lot of data, all very quickly, all in parallel (lots of tour groups flying in the same direction). To get high performance, your program needs enough parallelism to keep the thread processors busy (lots of customers). Each of your thread blocks needs enough parallelism to fill all the thread processors in a multiprocessor (big tour groups), and at least as many thread blocks as you have multiprocessors (many big tour groups). In addition, you need even more parallelism to keep the multiprocessors busy when they need to thread-switch past a long latency memory operation (many many big tour groups).

However, lots of parallelism isn't quite sufficient. The GPU is designed for structured data accesses, which graphics processing naturally uses (here's where my analogy falls apart). The memory is designed for efficient access to contiguous blocks of memory, meaning adjacent threads using adjacent data. Your program will run noticeably slower if it does random memory fetches, or (shades of vector process-

ing) simultaneous accesses to the same memory bank. But if your program is structured to use contiguous data, the memory can run at full bandwidth.

The program model reflects this structure as well. Consider a simple graphics shader; you want to compute the color at each pixel on the screen. Generally, the color at each pixel is more or less independent of the color at every other pixel (natural parallelism), and the problem has a natural index set (the two-dimensional pixel coordinate). So the program model is a scalar program to be executed at each pixel coordinate, replicated and parallelized over the index set. The scalar program is a kernel and the index set is the domain. The indices for each instance of the kernel determine which pixel is being processed, and are used to fetch and store data.

When you write a general purpose program for a GPU, you must currently follow this model as well. You must split out the computational kernels and the corresponding domains. You have to identify which data will live in the local memory, and move data between the local and device memory, keeping in mind that the local memory data only has a lifetime of a thread block.

Each kernel will complete its entire domain before the next kernel starts. In our airline analogy, you have to complete the Eastbound flight for all the tour groups on board before starting the next flight Westbound. The parallelism comes from the domain of a single kernel (data parallelism, all tour groups flying East at the same time), not from running many different kernels at once (task parallelism, lots of tour groups flying all over the country or world at once). This naturally limits (or focuses users on) the applications for which a GPU is appropriate.

Another key difference between GPUs and more general purpose multicore processors is hardware support for parallelism. GPUs don't try to address all the possible forms of parallelism, but they do solve their target range quite well. We've already mentioned the SIMD instruction sets within a multiprocessor and hardware multithreading. There is also a hardware thread control unit that manages the distribution and assignment of thread blocks to multiprocessors. There is additional hardware support for synchronization within a thread block. Your common multicore processor depends on software and the OS to provide these features, so advantage GPU.

Programming GPUs Today

So far I haven't really discussed how these highly parallel GPU architectures are programmed. Past accelerators were often programmed by offloading functions or subroutines; the user or compiler would marshall the arguments, send them to the accelerator, launch the function on the accelerator, and wait for completion, perhaps doing other useful work in the meantime. GPUs don't fit this model; they aren't fully functional, separately programmable devices. They really can only execute kernels, comprising a scalar kernel program and an index domain over which to apply it. If your function were that simple (matrix multiplication, SGEMM), it could look like a subroutine engine. Anything more complex, and the host has to manage the GPU execution by selecting and ordering a sequence of kernels to execute, and performing any scalar operations and conditionals along the way. In the next article, I'll discuss the sometimes superhuman efforts necessary to decompose a program into kernels and manage the data movement between the host and the GPU.

Compilers and More:
Programming GPUs Today

In the not-too-distant past, ENIAC was programmed with switches and a plugboard. Stored program computers soon followed that allowed one to write a program, load it into the computer memory, and run it. Initially, those programs had to be written in or manually translated into binary machine code, but soon assembly languages and assemblers were developed to simplify the process.

Soon followed operating systems, multiprogramming, and the concept of an application binary interface (ABI). The ABI defines the interface between an application and the operating system, libraries, and other components. One aspect of an ABI is to define a calling convention, including how arguments are passed to a function and where the return value can be retrieved. For instance, the x64 ABI defines that the first six integer or pointer arguments are passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9), the first eight floating point arguments (single or double precision) are passed in SSE registers (%xmm0 to %xmm7), and any remaining arguments are pushed on the stack (in right-to-left order). This allows up to 14 arguments to be passed in registers, which surely captures most function calls.

But not all; WRF, the Weather Research and Forecasting Model, is used for both atmospheric research and numerical weather prediction. A version of WRF is included in the SPEC CPU2006 suite. One routine (copying from the WRF source code) is "a mixed phase ice microphysics scheme" WSM5, with 49 arguments; it calls a subroutine WSM52D to handle the two-dimensional physics with 47 arguments (19 integers, 18 floating point scalars, 10 floating point arrays). Imagine writing the routine call by hand in assembly language; it takes over 100 instructions just to marshall the arguments and put them in the right place.

Instead, the computing community created higher level programming languages. While the first compiler (for the A-0 system) was more like what we would today call a loader, programming languages and compilers have progressed to where we use many higher level languages (C, Java, Fortran, others too many to enumerate) with a great increase in productivity. Much programming is done without using a textual language at all; for instance, a spreadsheet is a form of a program, and various visual programming interfaces exist. Now, the routine call in WRF with 47 arguments takes one Fortran statement, much easier to write and maintain than the corresponding assembly code:

```
CALL wsm52D(t, q(ims,kms,j), qci, qrs,              &
     w(ims,kms,j), den(ims,kms,j),                  &
     p(ims,kms,j), delz(ims,kms,j), rain(ims,j),    &
     rainncv(ims,j),delt, g, cpd, cpv, rd, rv, t0c, &
     ep1, ep2, qmin,                                &
     XLS, XLV0, XLF0, den0, denr,                   &
     cliq, cice, psat,                              &
     j,                                             &
     ids, ide, jds, jde, kds, kde,                  &
     ims, ime, jms, jme, kms, kme,                  &
     its, ite, jts, jte, kts, kte )
```

If programming in binary is akin to using fingers and teeth, and assembly language is like using sticks and stone knives, think of higher level languages as the power tools of programming.

Enter GPUs

The earliest GPUs were hardware graphics accelerators to handle line drawing, area fill, image transfer, and so on, offloading the CPU. The adoption of standardized libraries such as OpenGL and Direct3D drove the development of hardware 3D graphics accelerators, particularly with programmable shading capability. Since 2000, the programmability of the graphics accelerator chips has improved to the point where they can be used for nongraphics applications. This has been called GPGPU (Generate Purpose computation on GPUs). Early GPGPU programming used the existing graphics libraries, such as OpenGL, mapping between computing concepts (array, loop, execute) and graphics concepts (texture, kernel, draw). This is truly heroic programming. It's like using a chain saw to carve blocks of ice: in the right hands, it can produce something beautiful, but one wrong move and all you have is ice cubes (or worse).

More recently, the programming research and development community have tried to come up with programming models that would map well onto GPUs and similar parallel computers, particularly stream programming, evidenced in several projects: StreamIt at MIT, Sh at

Waterloo (which led to RapidMind), Brook at Stanford (which spun out briefly as PeakStream, and which AMD has adopted and extended as Brook+), and others.

The GPU programming model that has caught the most attention is NVIDIA's CUDA. The language is an extension to C; the software includes compiler, libraries, and many examples. There is a large user community, including a few dozen universities which use it in course work. Moreover, the software is free, though it (obviously) only targets NVIDIA GPUs.

Another programming model, very similar to CUDA and being sponsored by Apple and others, is OpenCL. The programming models are so similar that I'll only point out the differences here.

My last article discussed the GPU architectures and some of the problems facing programmers who want to compute on one. The first is to have an application with enough of the right type of parallelism to map onto the GPU. As the most parallelizable simple example, let's see what it would take to port a matrix multiplication to the GPU.

In Fortran, a matrix multiplication looks like a triply-nested loop:

```
do i = 1,n
  do j = 1,m
    do k = 1,p
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

In C, we have to decide whether to store the arrays linearized in a long vector, or whether to use a vector of pointers, or whether we have the degenerate case with fixed size arrays. Let's assume we use linearized arrays:

```
for( int i = 0; i < n; ++i )
  for( int j = 0; j < m; ++j )
    for( int k = 0; k < p; ++k )
      a[i+n*j] += b[i+n*k] * c[k+p*j];
```

Matmul is a wonderful example to use when experimenting with loop optimizations, because it can be rewritten in so many ways. The three loops can be interchanged or reordered in six ways, strip-mined or tiled, parallelized and vectorized. To optimize for vector instructions, we want the i (stride-1) index innermost, to maximize the memory fetch/store bandwidth. For parallel multiprocessor or multicore execution, we want the j index outermost, so each processor or core is computing distinct columns of a. To optimize for cache memories, we want to tile all the loops, so the innermost loops compute a submatrix multiplication where the submatrices all fit in cache. An optimized, parallelized, vectorized matmul for a quad-core processor might look like:

```
jts = j tile size;
ts = i tile size;
kts = k tile size;
parfor( int p = 0; p < 4; ++p )/* parallel loop */
  for( int jt = p; jt < m; jt += 4*jts )
    for( int it = 0; it < n; it += its )
      for( int kt = 0; kt < p; kt += kts )
        for( int j = jt; j < min(m,jt+jts); ++j )
          for( int k = 0; k < min(p,kt+kts); ++k )
            for( int i = 0; i < min(n,it+its); ++i )
               /* vector mode */
                a[i+n*j] += b[i+n*k] * c[k+p*j];
```

So, even optimizing this for a modern parallel workstation or server takes significant work, knowledge of the memory hierarchy, and experimentation. In the past, programmers would have to do this all manually, though advanced compiler technology is now able to achieve this kind of optimization automatically. However, we want to compute the matmul on the GPU, using CUDA. Let's list the steps we must take in our program to get there.

Initialize the GPU. since we only have to do this once for the whole application, I'll ignore this step.

Allocate memory on the GPU. We've already allocated the memory (explicitly or implicitly) on the CPU for the arrays, but the GPU executes from its own separate memory. So, the first thing we must do is allocate memory for new copies of the data on the GPU. In concept, it's just like executing a malloc on the GPU, but things are never quite so simple. We can start by simply allocating linear GPU memory:

```
cudaMalloc( &dev_a, n*m*sizeof(float) );
cudaMalloc( &dev_b, n*p*sizeof(float) );
cudaMalloc( &dev_c, p*m*sizeof(float) );
```

However, we may decide (or find) that the matrix columns aren't aligned on 64-byte boundaries (we're using column-major storage in our example). Since aligned memory accesses are faster than unaligned, we can force alignment by using a different allocation routine:

```
cudaMallocPitch( &dev_a, &pitch_a, n*sizeof(float), m );
cudaMallocPitch( &dev_b, &pitch_b, n*sizeof(float), p );
cudaMallocPitch( &dev_c, &pitch_c, p*sizeof(float), m );
```

This returns the allocated (aligned) size for the first dimension (the pitch), given the requested sizes of the two dimensions. There is a third option, allocating CUDA Arrays and mapping them into textures, which I'll not discuss.

Move data to the GPU. The b and c matrices on the host must be copied from host memory to GPU memory. Even though our example loops don't actually initialize the value of a to zero, we'll assume that's what we wanted, so we only have to move b and c. The actual data movement will be done with a hardware DMA transfer. Hardware DMA doesn't know about virtual memory and is optimized to move large contiguous chunks of memory across the PCI bus. We can ignore that issue and just move the data with a specialized memcpy call:

```
cudaMemcpy2D( dev_b, pitch_b, b, n*sizeof(float),
   n*sizeof(float), p, cudaMemcpyHostToDevice );
cudaMemcpy2D( dev_c, pitch_c, c, p*sizeof(float),
   p*sizeof(float), m,cudaMemcpyHostToDevice );
```

The arguments give the destination pointer and pitch, the source pointer and pitch, the two dimension sizes, and copy direction. If we want to optimize the data transfer, we can allocate the host arrays in page-locked (pinned) memory. This ensures the arrays don't get paged out by the virtual memory manager. The disadvantage is that pinning large amounts of memory reduces the amount of memory available for paging, potentially reducing performance for other applications running at the same time. CUDA provides handy routines to allocate and free pinned host memory. OpenCL seems to provide the ability to allocate and copy data in a single function call.

Select the kernel domain. As I mentioned last time, the GPU actually executes a (usually small) scalar kernel program on each point of a multidimensional domain. The selected domain affects both the host program (a little) and the kernel program (a lot). Moreover, the domain determines how much of what kind of parallelism is being used. I'm going to expand on this point more in my next article, but for now let's assume we've chosen to execute the i and j loops in parallel. This gives us a kernel domain of nxm, where the body of the kernel is the k loop.

Write the GPU kernel. Again, I'll expand on this next time around, but the kernel might look like:

```
__global__ void mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
  int i = blockIdx.x*32+threadIdx.x;
  int j = blockIdx.y;
  float sum = 0.0;
  for( int k = 0; k < p; ++k )
    sum += b[i+pitch_b*k] * c[k+pitch_c*j];
  a[i+pitch_a*j] = sum;
}
```

Recalling the last article, the kernel will run 32 copies in SIMT mode, with (n/32)xm thread blocks executing in parallel on the various GPU multiprocessors. Note: this is a particularly unoptimized matrix multiplication kernel, but it should work.

Run the kernel. Here is one point where the CUDA model really shines. Running the kernel actually takes several steps, which are nicely hidden by the CUDA compiler. The steps include:

1. Load the kernel to the GPU. When we run a program on your CPU, we depend on the operating system to load our program and prepare it for execution. If we use explicit shared objects or dynamic load libraries, our program will search for the object and load it at run time. GPU kernels use a similar model; the GPU kernel must be downloaded by the application from the host to the GPU. One significant benefit is the GPU kernel is typically stored in a portable format; when it is downloaded, it is translated and optimized for the specific GPU installed in the system. This lets us run the same program on systems with different GPUs, without having to recompile or reoptimize the application. Since the GPU manufacturers come out with new models every 9-12 months, this works to our advantage.

2. Define the execution domain; we've already decided on the domain, now we have to put it in the program.

3. Pass the arguments to the kernel from the host.

4. Launch the kernel. The execution can proceed asynchronously with the host, and the host can test whether the kernel has completed across its whole domain. In CUDA, this is done with a few lines:

```
dim3 threads( 32 );
dim3 grid( n/32, m );
mmkernel<<< grid, threads >>>( dev_a, dev_b, dev_c,
            pitch_a, pitch_b, pitch_c, n, m, p );
```

The NVCC compiler translates this into the steps outlined above. OpenCL is not so convenient. Since it is library-based, it can't depend on a compiler to simplify the steps. Instead, we will have to do each step explicitly, something approaching:

```
/* program is a prebuilt kernel program */
kernel = clCreateKernel( program, "mmkernel" );
grid[0] = n;
grid[1] = m;
threads[0] = 32;
threads[1] = 1;
/* context is the GPU compute context */
range = clCreateNDRangeContainer( context, 0, 2, grid,
       threads );
clSetKernelArg( kernel, 0, dev_a, sizeof(dev_a), NULL );
clSetKernelArg( kernel, 1, dev_b, sizeof(dev_b), NULL );
clSetKernelArg( kernel, 2, dev_c, sizeof(dev_c), NULL );
clSetKernelArg( kernel, 3, pitch_a, sizeof(pitch_a),NULL );
clSetKernelArg( kernel, 4, pitch_b, sizeof(pitch_b),NULL );
clSetKernelArg( kernel, 5, pitch_c, sizeof(pitch_c),NULL );
clSetKernelArg( kernel, 6, n, sizeof(n), NULL );
clSetKernelArg( kernel, 7, m, sizeof(m), NULL );
clSetKernelArg( kernel, 8, p, sizeof(p), NULL );
/* queue is a GPU work queue */
clExecuteKernel( queue, kernel, NULL, range, NULL, 0,NULL );
```

5. Wait for the kernel to finish. If we have a more complex computation, we might queue up several kernels; they will execute one after the other as they finish. In that case, we only have to wait until the last kernel is done.

6. Move results back from the GPU to the host. This, after all, is why we are doing the computation, to get the results. This is simply the inverse of loading data onto the GPU:

```
cudaMemcpy2D( a, n*sizeof(float), dev_a, pitch_a,
  n*sizeof(float), m, cudaMemcpyDeviceToHost );
```

7. Free the device memory. Again, the inverse of allocation:

```
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
```

CUDA (and, from reports, OpenCL) lets us program the GPU using a familiar language, C. However, a great deal of the programming is done using library calls, and the computation itself, the matrix multiplication, is divided into two parts: the kernel, on the GPU, and its invocation on the host. What was a multi-dimensional loop, which modern compilers are pretty good at optimizing, has turned into dozens of lines of code to manage memory, move data, and deal with the architectural specifics of the GPU.

I'm sure some (or most, or perhaps all) of the readers will at this point say "You shouldn't be programming matrix multiplication anyway; just call a library routine." That's true; in fact, NVIDIA provides versions of the BLAS routines, including SGEMM, which give very good performance. But, if it takes this much effort to get a matrix multiplication moved to the GPU, imagine how much effort it takes to move a real computation (say, WRF). Here I had only three arrays with regular access patterns and full freedom to parallelize the loops. Can you begin to see the difficulties?

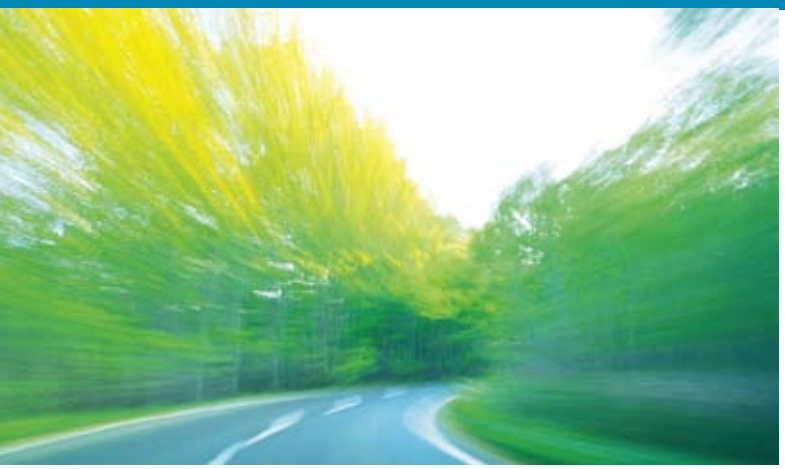
And I haven't even begun the real programming process, such as handling error returns from the runtime calls. Given the CUDA or OpenCL code, how portable will it be, how will I maintain it to keep it at the peak of efficiency?

As I mentioned above, the CUDA NVCC compiler simplifies some of the coding details. Compilers are good at bookkeeping, organizing details about memory addressing, alignment, and so on. OpenCL seems to be a step backwards, from a compiler-oriented solution aimed at raising the level of programming to a library-oriented solution aimed at giving low-level control to the programmer.

Low-level control isn't bad; that would be like saying assembly language is bad. It isn't bad, but it's only necessary for a very small bit of the programming we do today. If high level languages are the power tools of programming, we seem to be taking a step back to hand drills and saws. Many woodworkers prefer hand tools, and they can make beautiful furniture, but the cost is high and productivity is low. We need hand tools, but we'll be much more productive with better power tools.

In the next article, I'll look at the matrix multiplication kernel, exploring various ways to optimize for parallelism and memory bandwidth, and presenting performance. Whether you think programming the host side of a GPU program is hard, or just work, you'll be entertained, enlightened, or frightened when you see what goes into the GPU side. I'm on record as saying that parallel programming isn't easy and never will be, but we can and should develop the tools and training to turn this Acceleration Nightmare into something about as scary as a Halloween Haunted House.

Compilers and More:
Optimizing GPU Kernels



My last article discussed some of the complexities of programming GPUs today, focusing on how to interface the host program with the GPU. Here we focus on programming the GPU itself. As with last time, we'll look at a simple single-precision matrix multiplication, equivalent to the BLAS SGEMM routine.

Matmul is a highly parallel algorithm, but let me emphasize that parallelism does not equate to performance. We need to carefully sculpt our algorithm to match the parallelism available in the architecture in order to reap the benefits. This is true whether we are targeting a GPU, a multicore x64, or even a single core with packed SSE operations. As an example, I took the simple matmul loop (in C, but with the matrices stored column-major):

```
for( int j = 0; j < m; ++j )
  for( int k = 0; k < p; ++k )
    for( int i = 0; i < n; ++i )
      a[i+pitch_a*j] += b[i+pitch_b*k]*c[k+pitch_c*j];
```

modified it several ways and ran it on an Intel Xeon (3GHz, 6MB cache, 16GB memory, Penryn) using 4096x4096 matrices (to compare with results we'll see below). With the loop in the order shown (stride-1 inner loop), the program ran at 1.7 GFLOPs; this is compiled C performance (using pgcc -fast). We can improve that by tiling or blocking the loops, organizing the matmul as a bunch of submatrix multiplications, sized so each submatrix matmul fits in the processor cache. This improves performance to 5.7 GFLOPs, and it jumps to over 22 GFLOPs when we use OpenMP directives and run on all four cores. Advanced compilers help by automatically managing the vectorization, unrolling, memory alignments, adding prefetch instructions, and so forth.

We're going to see several matmul GPU kernels, with performance on our GPU development system, with an NVIDIA GeForce GTX 280 (1GB memory, 30 multiprocessors), using NVIDIA's CUDA language. The host is a Linux (OpenSUSE 11.0) triple-core AMD Phenom (2.1 GHz, 500KB cache, 4GB memory), though the host hardly matters; the performance for these experiments is entirely dominated by the GPU code.

As on the CPU, performance on a GPU can be fragile; small changes to the program can make large differences in performance. It's easy to write a slow program. This was a characteristic of High Performance Fortran, one that (my opinion) was a major cause of its downfall; while HPF made it easier to write parallel programs, it didn't make parallel programs fast. That is the job of the HPC programmer; the same will be true for accelerators, GPUs, and even multicore CPUs.

GPUs deliver their dramatic high performance through a well-balanced, carefully managed, highly parallel architecture. Algorithms running on the GPU must be parallelized and balanced as well; this does not come for free. Program development may cost extra time and effort to understand and use the appropriate programming model, a model that may not match the simple scalar processor with cache model we are comfortable with on x64 hosts. However, the analysis and programming techniques used to develop GPU algorithms will probably help you develop multicore programs as well. A good programming model with good compilers and tools can relieve you of much busywork, but you still have to think, and you still have to

understand algorithms and architecture, and you should expect no less.

From here on below, I show many versions of matmul; if you're not a programmer or want to skip over the details, look for the performance tags below, until the Summary; you don't want to miss the conclusions. If you are a programmer and want to see all the code, you'll find all the sources in a kernels tarfile on the PGI website at www.pgroup.com/lit/kernels/kernels.tar.

In the last article, I proposed a simple matmul kernel for the GPU and focused on the host code to drive the kernel. We'll use that simple kernel to start the discussion. What I had done is taken the matmul loop (as shown above), strip-mined the stride-1 i loop to the CUDA SIMD width of 32:

```
for( int is = 0; is < n; is += 32 )
  for( int i = is; i < is+32; ++i )
    for( int j = 0; j < m; ++j )
      for( int k = 0; k < p; ++k )
        a[i+pitch_a*j] += b[i+pitch_b*k] * c[k+pitch_c*j];
```

run the i element loop as a thread block, and run the is strip loop and j loop in parallel:

```
parfor( int is = 0; is < n; is += 32 )     /* K1 */
  parfor( int j = 0; j < m; ++j )
    SIMDfor( int i = is; i < is+32; ++i )
      for( int k = 0; k < p; ++k )
        a[i+pitch_a*j] += b[i+pitch_b*k] * c[k+pitch_c*j];
```

then optimized by hand just a little. The parallel grid loops and the SIMD thread block loop are handled implicitly by the GPU hardware and firmware, so they don't appear in the kernel code. All that's left is the body, the k loop. The final kernel in all its glory, cut-and-pasted from my CUDA source file, is:

```
extern "C" __global __void    /* K1 */
 mmkernel( float* a, float* b, float* c,
   int pitch_a, int pitch_b, int pitch_c,
   int n, int m, int p )
 {
     int i = blockIdx.x*32 + threadIdx.x;
     int j = blockIdx.y;
     float sum = 0.0;
     for( int k = 0; k < p; ++k )
       sum += b[i+pitch_b*k] * c[k+pitch_c*j];
     a[i+pitch_a*j] = sum;
  }
```

28 GFLOPs - SIMD width 32

This version runs at 28 GFLOPs on our system (on 4096x4096 matrices). In the interest of full disclosure, I compiled the kernels discussed here with NVIDIA's NVCC compiler version 2.0 with the -O option, and compiled the driver routine with pgcc -fast; I ran each program three times and report the middle performance score, rounding the GFLOPs down to an integer value. I will generally show a GFLOPs number most directly comparable to a host matmul, including the overhead of transmitting the operand matrices to the GPU memory and the result matrix back. I will sometimes give the performance of just the matmul kernel on the GPU; while the two numbers are often quite close, the kernel-only number is useful to expose more clearly the effect of changes to the kernel program (since the overhead stays the same). I show results for 4096x4096 matrices, which is close to the peak performance for each kernel. For version K1, the host-to-host and kernel-only performance were 28 and 29 GFLOPs, respectively.

That may sound like good performance, but we're not nearly taking full advantage of the available parallelism. Recall the NVIDIA architecture description; the card I'm using has 30 multiprocessors, each with eight thread processors, quad-clocked to get a SIMD width of 32. The kernel above is a scalar program, but the card runs 32 copies of it in SIMD mode (or SIMT mode, to use NVIDIA's term); the 32 copies comprise a warp. Each multiprocessor uses multithreading to support up to 32 warps (1024 scalar threads). The 32 warps can come from different thread blocks (different iterations of is or j) or from wider thread blocks (more than 32 scalar threads). There are limits in this

generation of the card: each multiprocessor can support up to eight simultaneous thread blocks, and a thread block can support up to 16 warps.

36 GFLOPs - SIMD width 64

The K1 kernel has only one warp per thread block, so at most eight thread blocks will be active on each multiprocessor, out of the possible 32. I can improve that by strip-mining the i loop to a width of 64, changing the 32 to a 64 in kernel K1, and running with 64 threads in each thread block. With this version, if eight thread blocks are scheduled on each multiprocessor, we get up to 16 warps, so the multithreading is more effective. And we see a performance increase, to 36 GFLOPs (38 kernel-only).

35 GFLOPs - SIMD width 128

So what happens if I try this trick again, doubling or quadrupling the strip size to 128 or 256? This increases the maximum number of warps per multiprocessor to 32 (which is the limit), so we might expect another bump in performance from improved multithreading. Unfortunately, we don't; the performance drops slightly to 35 GFLOPs (36 kernel-only) in both cases. This I can't quite explain.

1.7 GFLOPs - SIMD width 32, non-stride-1 array accesses

Even with this simple version, I made some assumptions and optimizations, knowing something about the machine. I know that stride-1 accesses in a thread block are important, so I ran the stride-1 i loop along the thread index. Just how important is that? Suppose we switch the i and j indices, so the SIMD memory accesses are along a column; the performance drops from 28 GFLOPs (K1) to 1.7. We can call this kernel Ks (s for stupid, or slow).

Still, we've only just started. If we inspect the code for kernel K1, we note that the inner loop contains two memory fetches, for b and c; both fetches are from the device memory, which has a very high latency. In particular, the fetch for c loads the same element for all the threads in the thread block. The memory system is designed for high bandwidth when all the threads access consecutive elements, such as with the b access. This used to be called superword access in classical vector machines, where the memory returns 64-bytes (or more) at a time. Kernel K1 doesn't take advantage of this memory design for the c access, but we can fix that. Let's strip-mine the k loop, and load a strip of c into the multiprocessor local memory. The pseudo code is:

```
parfor( int is = 0; is < n; is += 32 )     /* K2 */
  parfor( int j = 0; j < m; ++j )
    SIMDfor( int i = is; i < is+32; ++i )
      for( int ks = 0; ks < p; ks += 32 )
        cb[ks:ks+31] = c[ks+pitch _ c*j:ks+31+pitch _ c*j];
        for( int k = ks; k < ks+32; ++k )
          a[i+pitch _ a*j] += b[i+pitch _ b*k] * cb[k-ks];
```

33 GFLOPs - cached access to c, SIMD width 32

55 GFLOPs - cached access to c, SIMD width 64

63 GFLOPs - cached access to c, SIMD width 128

Note the vector fetch of c into the temporary array cb. This is handled in kernel K2 by letting each thread fetch one element and storing into the multiprocessor local memory, so the inner loop only has one device memory fetch; the fetch of cb from the local memory is almost as fast as a register access; we see the performance improves to 33 GFLOPS, up from 28. We can again increase the number of threads per block from 32 to 64 and 128, and we see performance improve from 33 to 55 and 63 GFLOPs. As with kernel K1, increasing to 256 threads per block does not improve performance. An implementation detail: with more than one warp per thread block, we need to synchronize the warps after loading the temp array cb, and before reloading it the next time around the ks loop; see the CUDA source code for this detail.

63 GFLOPs - cached access to c, SIMD width 64, unroll inner loop

But we're not done yet. What if we unroll the inner loop, to reduce the loop overhead? We might unroll to a factor of 2 or 4 or even 16. Unrolling the inner loop once in the 64-wide K2 kernel does improve performance, getting 63 GFLOPs, but more unrolling doesn't help, and it doesn't help the 128-wide kernel.

So far we've got two kernel versions, with variations in the thread-block (vector) size and unrolling. And we've only just begun. We tried unrolling the inner k loop; what if we try unrolling one of the outer loops? We could let each kernel instance compute two values of the i loop. The pseudo-code looks like:

```
parfor( int is = 0; is < n; is += 64 )  /* K3 */
  parfor( int j = 0; j < m; ++j )
    SIMDfor( int i = is; i < is+32; ++i )
     for( int ks = 0; ks < p; ks += 32 )
      cb[ks:ks+31] = c[ks+pitch _ c*j:ks+31+pitch _ c*j];
       for( int k = ks; k < ks+32; ++k )
        a[i+pitch _ a*j] += b[i+pitch _ b*k] * cb[k-ks];
          a[i+32+pitch _ a*j] += b[i+32+pitch _ b*k] * cb[k-ks
```

53 GFLOPs - cached access to c, SIMD width 32, unroll i loop

63 GFLOPs - cached access to c, SIMD width 32, unroll i loop 3x

Each iteration of the i loop now computes values for i and i+32. We don't expect much advantage here, since the only values shared between the two i iterations are loaded from the local memory, which is already pretty fast. But even this kernel improves upon K2, with 53 GFLOPs. We can improve this to 63 GFLOPs by unrolling more or increasing the SIMD width to 64.

Next, we can try unrolling the j loop, so each kernel computes values for j and j+1. The pseudo-code is:

```
parfor( int is = 0; is < n; is += 32 )     /* K4 */
  parfor( int j = 0; j < m; j += 2 )
    SIMDfor( int i = is; i < is+32; ++i )
      for( int ks = 0; ks < p; ks += 32 )
        cb0[ks:ks+31] = c[ks+pitch_c*j:ks+31+pitch_c*j];
        cb1[ks:ks+31] = c[ks+pitch_c*(j+1):ks+31+pitch_c*(j+1)];
        for( int k = ks; k < ks+64; ++k )
          a[i+pitch_a*j] += b[i+pitch_b*k] * cb0[k-ks];
          a[i+pitch_a*(j+1)] += b[i+pitch_b*k] * cb1[k-ks];
```

59 GFLOPs - cached access to c, SIMD width 32, unroll j loop

98 GFLOPs - cached access to c, SIMD width 64, unroll j loop

117 GFLOPs - cached access to c, SIMD width 128, unroll j loop

Here, we note the two assignments in the k loop fetch the same value of b from the device memory. This version gets 59 GFLOPs; it jumps to 98 GFLOPs when we increase the SIMD width to 64, and again to 117 GFLOPs with a SIMD width of 128. Now we're starting to see real performance, over 100 GFLOPs, host-to-host.

176 GFLOPs - cached access to c, SIMD width 128, unroll j loop 3x

But we're not done yet. What if we unroll the j loop by four iterations instead of just two? This involves keeping four partial sums. Now the performance with SIMD width 128 is 176 GFLOPs host-to-host, and over 210 GFLOPs on the device.

202 GFLOPs - cached access to c, SIMD width 128, unroll j loop 3x and k loop 1x

208 GFLOPs - cached access to c, SIMD width 128, unroll j and k loops 3x

More unrolling of the j loop doesn't improve performance, but what if we combine this with unrolling the k loop? If we unroll the j loop 3 times and the k loop once, with SIMD width of 128, we get 202 GFLOPs; unrolling the k 3 times gives us 208 GFLOPs (host-to-host), and 265 GFLOPs (kernel-only).

Our peak performance so far looks pretty good. It took some experimentation, but we have a version that uses only 128 threads and 2KB local memory per thread block, allowing up to 8 thread blocks on each multiprocessor, so taking great advantage of the multithreading properties of the machine. We haven't even fully explored all the combinations. What if we combine i loop unrolling with the j and k loop unrolling? Should we explore other unroll factors as we combine unrolling multiple loops? What if we use pointer arithmetic instead of array references (this really questions whether NVCC optimizes the array references, but it seems to do a good job there)? I desperately wanted to break the 200 GFLOP barrier, and reached it. The version of SGEMM that comes with CUDA BLAS gets about 260 GFLOPs (host-to-host) on a 4096x4096 matrix; I've still got some work to do to get that extra 25%.

When we optimize a matmul for a general purpose CPU with a cache, we've learned that we need a tiled algorithm. We can do the same thing on the GPU, where we fit the submatrices in the local memory. The pseudo code is:

```
parfor( int is = 0; is < n; is += 16 )     /* K5 */
  parfor( int js = 0; js < m; js += 16 )
    SIMDfor( int i = 0; i < 16; ++i )
      SIMDfor( int j = 0; j < 16; ++j )
        /* init A tile */
        at[is:is+15][js:js+15] = 0.0;
        for( int ks = 0; ks < p; ks += 16 )
           /* load B tile */
           bt[i][ks:ks+15] = b[i+pitch_b*ks:
                              i+pitch_b*(ks+15):pitch_b];
           /* load C tile */
           ct[ks:ks+15][j] = c[ks+pitch_c*(js+j):
                             (ks+15)+pitch_c*(js+j)];
           /* tile MM */
           for( int k = ks; k < ks+64; ++k )
              at[i][j] += bt[i][k]*ct[k][j];
        /* store A tile */
        a[i+pitch_a*j] = a[i][j];
```

164 GFLOPs - tiled loops, SIMD width 16x16, cached access to b and c

We have to choose a tile size, and square tiles seem to make as much sense as any other shape, at least to start with. We choose 16x16 tiles and run 256 threads in a thread group, so each thread will compute one element of the a tile; this lets us keep that element in a register. The actual kernel code is slightly more complex than the previous kernels. It's important to recall that this scalar kernel is one of a thread group or cohort of 256 cooperating instances, and it only works in that domain. This version gives us 164 GFLOPs, not quite as good as we've already seen. Why not? One reason is the thread group is 256 threads, so we hit the 1024 threads/multiprocessor limit with only four thread groups. We can address that as well, but I still haven't quite reached the peak performance shown on kernel K4.

The CUDA blas (260 GFLOPs) SGEMM is similar to this tiled version. It's based on work by Vasily Volkov, a Computer Science PhD student at Cal; Vasily's code uses a 16x4 thread block with the i loop unrolled by 16, the j loop by 4, and the k loop by 16 (if I read it right). The code might be hard to follow, but it sure beats trying to code a matmul in DirectX.

So let's suppose we've decided on the K4 algorithm. It assumes that the matrix sizes are multiples of 32 (or 64 or 128) in all dimensions, though it doesn't require the matrices to be square. One way to satisfy this is to pad all your matrices, filling in zeroes in the extra rows and columns. Matrix addition and multiplication will preserve these zeroes and will not pollute the actual values; this may be your best option. Another solution is to add conditionals so as to not run off the ends of the matrices. This complicates the code and can affect performance. The simplest method to test for array limits is to put conditionals around the device memory fetch and store operations; if we fill in zeros to the b and c tiles, the innermost loop won't need any tests. I reproduce the body of the kernel here:

```
float sum0 = 0.0, sum1 = 0.0;
for( int ks = 0; ks < p; ks += 32 ){
  if( ks+tx < p && j < m )
    cb0[tx] = c[ks+tx+pitch_c*j];
  else
    cb0[tx] = 0.0;
  if( ks+tx < p && j+1 < m )
    cb1[tx] = c[ks+tx+pitch_c*(j+1)];
  else
    cb1[tx] = 0.0;
  __syncthreads();
  if( i < n ){
    for( int k = ks; k < ((ks+32 < m) ? ks+32 : m); ++k ){
      float rb = b[i+pitch_b*k];
      sum0 += rb * cb0[k-ks];
      sum1 += rb * cb1[k-ks];
    }
  }
  __syncthreads();
}
if( i < n && j < m )
  a[i+pitch_a*j] = sum0;
if( i < n && j+1 < m )
  a[i+pitch_a*(j+1)] = sum1;
```

Even if i and j are outside the matrix bounds, we can't just skip the body of the loop for two reasons. First, each thread is part of a thread group, and as such it loads part of the data into the local temporary arrays cb0 and cb1; even if this thread has nothing to compute, it has to do its part of the shared work. Second, we have those pesky barrier synchronizations; all threads in a thread group must participate in the barrier, so even if this thread has no work to do, it had better reach those barriers.

These tests cost about 5% in performance, in the simplest version of K4. It's less costly in the more complex versions, but the code gets messy when mixed with some of the unrolling. But it will work with any matrix size, whereas K4 requires the size to be a multiple of 32.

Of course, if you need to deliver a library that works regardless of the matrix sizes, you have another option. You can create two versions of your routine, a faster one that works when the matrix sizes are appropriate multiples of 16, and a slower, general purpose one that works for other matrix sizes, with a conditional test to execute the right one. Then you get your good benchmark numbers (all benchmarks use large powers of two, right?), and you get right answers, too.

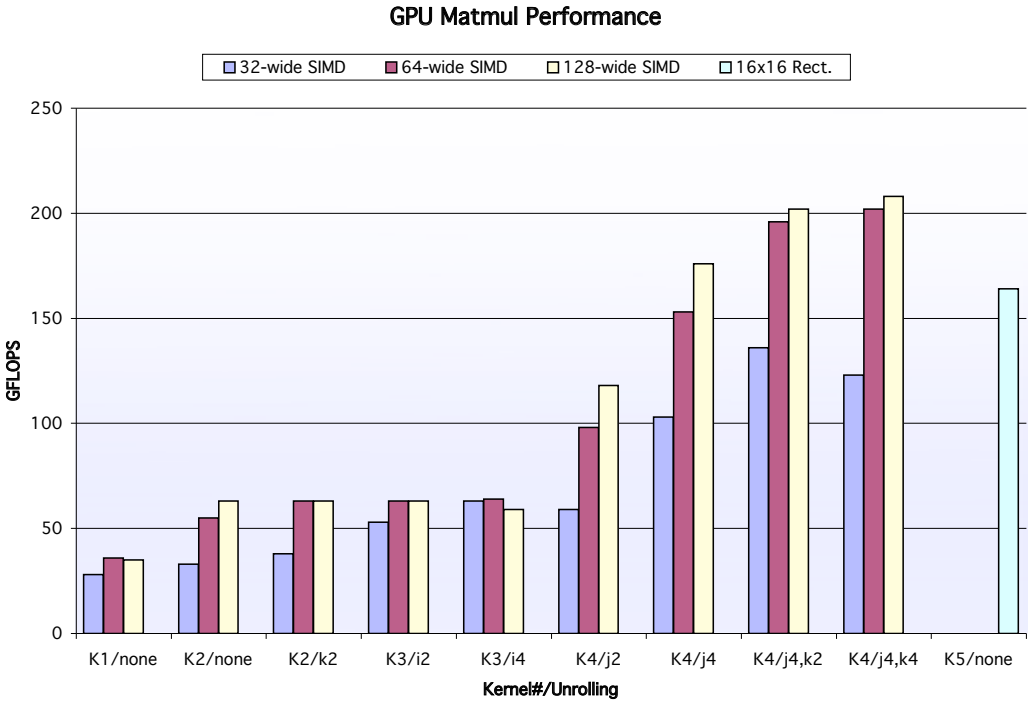
Summary

The point I've tried to make is how sensitive the performance of the GPU is to the formulation of your kernel, and how much and what kind of experimentation you'll need to do to optimize your performance. How much of the optimization process will carry over from one GPU to another, or from one generation to the next from the same vendor? Many programmers like this low level of control, and it certainly could be appropriate when developing a numerical library, in the same way assembly language is appropriate.

To be fair, the same is true on your CPU as well; you need to optimize your matmul for (packed) vector operations, memory strides, and cache locality. A bad program will run several times slower than a good one; ordering the matmul loops so the inner loop is non-stride-1 reduces the performance on large matrices (on our Penryn) by more than a factor of 10. But compilers and tools are far more mature and helpful when compiling for an x64, IBM POWER, Sun SPARC, or other CPU.

I'm sure many readers would like to tell me (again) that I should be using the prebuilt library version of SGEMM for matmul, not writing my own. Save your breath. Matmul is just one simple example here, three loops, three matrices, lots of parallelism, and yet I put in several days of work to get this seven line loop optimized for the GPU.

We can compare the evolution of GPU programming to the evolution of shared-memory parallel programming.



"I desperately wanted to break the 200 GFLOP barrier, and reached it."

There were many new languages designed to take advantage of parallelism (Id, SISAL, many others). Many low-level libraries were written to create and manage threads, eventually standardizing on POSIX threads (aka pthreads). Much work was done on automatic parallelization, dating back to the 1960s and 1970s. When successful commercial shared-memory multiprocessors became more widely available in the 1980s, an effort began to standardize a less intrusive programming interface for multiple processors, driven by multiprocessor workstations, eventually resulting in the OpenMP API, which defines directives and a runtime interface to a shared-memory parallel programming model.

GPUs have their own set of domain-specific languages, including GLSL (OpenGL Shading Language), HLSL (high level shader language) from Microsoft for DirectX, and Cg (C for graphics) from NVIDIA. We're now in a period with development of low-level libraries and interfaces to create and manage GPU threads; the OpenCL effort aims to standardize this. The cost to port a nontrivial application to this model is high, though the potential performance is alluring.

Luckily for me, my application (the compiler) runs on the host, and I don't have to port that. But what about the real application programmer, who has thousands (or hundreds of thousands) of lines of code? Is it feasible to take GPU or accelerator programming concepts, abstract them into a predictable and useful programming model, and present them using a portable programming interface, in the same way that OpenMP abstracts and presents multiprocessor and multicore systems? That's a topic for the next article.

Compilers and More:
A GPU and Accelerator Programming Model

Okay, maybe the title should be "Languages and More," but I promise to talk about compilers further on below.

In these articles I've discussed parallel programming, and programming GPUs and accelerators in particular. In an earlier HPCwire article, I predicted that accelerator-based systems would dominate high performance computing, and suggested that an evolutionary approach to migrating applications from CPUs to accelerators was possible and appropriate. In the first article here, I discussed in more detail the specifics of GPU hardware architecture, pointing out its strengths for high performance computing (lots of parallelism), as well as its weaknesses (limited to specific parallelism domains). In the second article, I showed what it takes to start porting a CPU program to a GPU, exposing some of the complexities of the interactions between the host and the GPU. The specific examples use NVIDIA's very popular CUDA language, but I discuss OpenCL briefly as well. OpenCL should be about ready for public discussion by the time you read this. In the third article, I showed the details of optimizing a simple matmul kernel for a GPU, including testing various ways to organize it and vary the parallelism parameters.

If you read these, or are familiar with current approaches to programming accelerators, you are either discomforted by the complexities, or excited at the levels of control you can get. The low-level programming model in CUDA and OpenCL certainly has its place, though it's not for the faint of heart. So, to go back to the first of these articles, can we come up with a different model of GPU and accelerator programming? One that retains most of the advantages of CUDA or OpenCL, but without requiring complete program rewrites? That can be applied to different target accelerators, and that retains the potential to develop and test in a more accessible environment? In short, a model that allows HPC programmers to focus on domain science instead of on computer science?

Architectural Model

Let's start by looking at the features of the architecture that we want to use to advantage. Current GPUs are specific implementations of a programming model that works well for graphics problems. They support two levels of parallelism: an outer fully-parallel doall loop level, and an inner synchronous (SIMD or vector) loop level. Each level can be multidimensional (2 or 3 dimensions), but the domain must be strictly rectangular. The synchronous level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the doall level.

For those familiar with memory models, current GPUs implement a particularly weak model. In particular, they don't support memory coherence between threads, unless those threads are parallel only at the synchronous level and the memory operations are separated by an explicit barrier. Otherwise, if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. You can't say it gets the wrong answers, because such programs are defined as being in error. There is a software-managed cache on a GPU, and there are some hardware caches that can be used as well, but only in certain situations and limited to read-only data.

The most significant characteristic is that the memory on the GPU or accelerator is separate from the host memory. The host can't simply read or write to the accelerator memory because it's not mapped into the virtual memory space of the host. Similarly, the accelerator can't simply read or write to host memory; the host memory doesn't support the bandwidth necessary for the accelerator, not to mention the need to support the virtual memory map on the accelerator.

The chips support parallelism on the order of hundreds of threads today, but effective programs need parallelism on the order of thousands. This provides enough slack parallelism to tolerate long latency memory operations by thread switching, or multithreading, an idea pioneered by the venerable Denelcor HEP almost 30 years ago.

In summary, today's GPUs look like an attached processor with its separate memory, that supports a multidimensional rectangular domain of parallelism, including doall and synchronous parallelism. We'd like a programming model that simplifies most of the hardware details, but gives experts finer levels of control. We probably can't hide the distinction between the two levels of parallelism, but we'd like to avoid requiring the programmer to insert explicit synchronization as much as possible. It's easy to map doall parallelism onto SIMD parallelism, but not the other way around, so we'd like to encourage programmers to program in a doall style when possible and appropriate. We probably can't completely hide the distinction between host memory and accelerator memory, but the details of transferring data should be handled automatically.

Our accelerator programming model shouldn't focus on the details of today's GPUs as the ultimate accelerator architecture. One can envision accelerators with mostly (or only) synchronous parallelism (like the Clearspeed CSX700 accelerator processor), or with mostly doall parallelism (like the Tilera TILE64 chip). Future accelerators may share physical and/or virtual memory with the host, and may support a stronger memory model with richer synchronization methods. Software and hardware cache architectures are likely to change rapidly. A robust programming model should express parallelism broadly enough that compilers and tools can map an application onto future generations of accelerators as well as it does onto today's GPUs. In fact, a successful model should be able to map applications onto a multicore X64 processor, where the SSE instructions implement the synchronous parallelism, and the doall parallelism is mapped across cores. From the available details, this model would even map well onto Intel's proposed Larrabee chip. There will be work to tune the performance for each architecture, both in the tools and even at the application level, but the parallelism model needs to be reasonably portable.

Programming Model

How should we implement an accelerator-targeted programming model? Three options immediately come to mind: library, language, or directives. Looking at the array of parallel programming choices, all intended to make parallel programming easy, they span all three options.

Library-based solutions are attractive for many such problems; they are easy to port and can be independent of processor or compiler vendor. The MPI communication library for large system communication is one well-known example. It's often easier to create and modify a standard for a library than for a language.

Language-based solutions expose the semantics in the language, allowing compilers or other tools to analyze and optimize the program. Co-Array Fortran, which is (currently) part of the next (allegedly minor) revision of the Fortran standard, exposes MPI-like parallelism and communication in the language, similar in some respects to Unified Parallel C (UPC). A compiler for Co-Array Fortran might be able to discover that a data copy from one image (thread) to another in a loop could be vectorized, given the appropriate support in the communication layer; such analysis in an MPI program is left entirely to the programmer. However, languages are expensive to implement, typically change quite slowly, and mistakes are hard to remedy once the standard is written.

A directive-based approach has some of the advantages of language-based solutions, in that directives expose the semantics to the compiler and other tools, allowing intelligent analysis and optimization. Such an approach

also allows a program to be developed and tested on platforms that don't support the directives, since the base language is unchanged. OpenMP is a widely available, successful parallel programming model based on directives to describe the parallel regions of the program.

Getting good performance on today's accelerators depends on selecting a region that has enough work to amortize the overhead of moving data between the host and accelerator. This is one instance of the more general problem of selecting a region that has enough compute intensity to amortize the data traffic across the memory hierarchy, be it separate memory or multilevel caches. Someday, we may trust compilers to make this determination automatically, but not yet.

So, let me propose a model that borrows strategies from OpenMP, since I'm the PGI representative to the OpenMP group. I'll propose directives in C and Fortran programs to delineate the regions of the program (loops) that should be accelerated (compiled for the GPU or other accelerator). Since the architecture model uses regular rectangular domains, I'll propose using parallel loops as the primitive parallel operation.

The keys to tuning are minimizing and perhaps optimizing the data traffic between the host and accelerator, and selecting a schedule for the parallelism. In many cases, a compiler can analyze the nested loops and determine the input and output data sets, so it can manage the data traffic automatically. However, we should never trust automatic analysis to solve all our performance problems, so we'll need directives or clauses to modify or augment the analysis.

As for scheduling, we saw in the previous article that there can be many possible schedules for even the simplest of parallel loops. Recent academic research in this area depends on doing more or less what I did by hand: generating many versions of the program and running each of them, then choosing the best one from the bunch; see Shane Ryoo's PhD dissertation (University of Illinois, 2008), and joint work from Professors Ramanujam and Sadayappan (Louisiana State University and The Ohio State University) as good examples.

Such an approach is valid for research, or when searching for a good algorithm for a highly tuned library, but inappropriate for a compiler. Instead, we will depend on the compiler to determine a reasonably good schedule (as we do when we use automatic parallelization and vectorization today), again with directives or clauses to modify or augment the decisions.

It's important that a programmer be able to control any compiler optimization decision here; the difference between good and bad performance is quite dramatic, and at least in the immediate future, any compiler decision will be made with only partial information. However, to support this requires that the compiler tell the programmer what decisions it has made, and hopefully why, so the programmer knows whether it's appropriate to step in and make a change.

So let me propose two directives. The first delineates an accelerator region, with optional clauses to control the data movement between host and accelerator memory. Borrowing liberally from OpenMP, I'll propose a `#pragma acc` prefix for C directives, and `!$acc` prefix in Fortran. In C, I'll describe an acceleration region as:

```
#pragma acc region
{
   /* loops to be accelerated go here */
}
```

Fortran doesn't have structured blocks (yet), so we'll use region and end region directives:

```
!$acc region
    ! loops to be accelerated go here
!$acc end region
```

Compare these to the OpenMP parallel regions. I propose optional clauses to tell the compiler what data needs to be copied into the region, from host to accelerator, what data needs to be copied out, and what data is local to the region; local data corresponds roughly to OpenMP private data. Compiler analysis is often able to determine the input, output and local data automatically.

The second directive is used to describe the mapping of parallel loops onto the hardware parallelism, what I called the schedule earlier. This corresponds roughly to the OpenMP loop directive, which describes the work-sharing pattern of parallel loops. It's probably easiest to explain with a familiar example; in the previous article, I showed several versions of matmul in CUDA with different schedules. The first (and simplest) version would be written (in Fortran) using these directives as:

```
!$acc region
    !$acc do parallel
    do j = 1, m
       do k = 1, p
          !$acc do parallel, vector(32)
          do i = 1, n
             a(i,j) = a(i,j) + b(i,k)*c(k,j)
          enddo
       enddo
    enddo
!$acc end region
```

The loop directives do two things: the first is to tell the compiler about loop-level parallelism, augmenting its analysis. The second is to tell the compiler how to schedule or map the loop-level parallelism onto the hardware. In this loop, both the i and j loops exhibit doall parallelism, but we want to map the stride-1 i loop onto the synchronous (vector) parallelism in strips of size 32, using doall parallelism between the strips. We expect compilers to issue a warning message if a programmer inserts a do parallel directive on a loop that compiler analysis shows is in fact not parallel. Compare this code for clarity with the corresponding CUDA kernel in the previous article.

This isn't intended to be a user guide, tutorial, even a formal proposal, but I hope to convince you that a directive-based approach is feasible in the short term, and can address many of the problems programmers will face when porting large applications for use on host+GPU platforms in particular, and host+accelerators in general.

This model does use reasonably sophisticated compiler analysis, but nothing that hasn't been implemented in commercial parallelizing compilers for many years. In this example, the compiler must take the following steps:

- Determine what data is input to the region; for this loop, the input data is a(1:n,1:m), b(1:n,1:p), c(1:p,1:m), and the loop limits.
- Determine what data is output to the region; this is simply a(1:n,1:m).
- Determine what data is local to the region, which is empty (except perhaps for the loop counters). Classical data flow and array region analysis solves all three of these problems.
- Determine which loops can run in parallel, augmented by information in the directives. For this loop, the j and i loops are completely parallel; the k loop requires a sum reduction, which is less efficient but could still be parallelized.
- Determine the loop schedule; in this example, the schedule is specified by the directives. Without the loop directives, the compiler would have to search among the possible schedules and select a best one; note to academics: this is still a fertile area for continued research.
- Generate code for the accelerator. For the most part, this is a classical compiler problem, and well known methods apply. On a target like the NVIDIA GPU, optimizing for the software-managed cache adds some complexity, but such problems have been addressed on past machines as well.
- Generate host code to move data to the accelerator, launch the accelerator kernel(s), and move results back from the accelerator.

Final Words

Will adoption and use of directives such as these make GPUs more generally applicable? These directives may make GPUs more accessible, but there are still serious limitations to the parallelism GPUs support. The restrictions include rectangular domains, two levels of parallelism, limited synchronization, and a weak memory model (in the formal sense). This makes it unlikely that anyone will be porting unstructured mesh code or dynamic pointer-chasing data structures to a GPU anytime soon.

Can this programming model be adapted to make parallel programming easy in general? I've argued that parallel programming is difficult, and always will be, regardless of the programming model, and I'm not backing down. To reiterate, this directive model is intended to make accelerator programming accessible, so programmers can focus on algorithms and performance, not on syntax and other trivialities.

This proposed style of parallel programming isn't universal, but it does address a significant segment of the parallel community. The model is portable, across GPUs, across accelerators, even to multicore CPUs, though we need to develop the compilers. Moreover, it's nicely incremental; you can use these directives to accelerate parts of your program without having to undertake a whole rewrite, and, as with OpenMP, you can still build and test your application on the host by ignoring the directives altogether.

Visit www.pgroup.com/resources/articles.htm to find links to all of Michael Wolfe's HPCwire articles.

The Portland Group

For more information about Accelerator Programming
www.pgroup.com