Technical News from The Portland Group

PGI Home Page

August 2009

The PGI Accelerator Programming Model on NVIDIA GPUs Part 2 Performance Tuning

by Michael Wolfe, PGI Compiler Engineer

The first installment of this series introduced the PGI Accelerator Programming Model, showing three simplfiee programs in C and in Fortran, and presenting a few details of building and running a program on the GPU. Here we discuss some issues affecting performance and how to recognize and address them. We'll discuss the four most important performance issues: writing an appropriately parallel algorithm,tuning the data movement between the host and the accelerator, tuning memory loads and stores on the accelerator, and tuning the loop schedule. For all of these, we'll need to understand how to interpret the compiler messages in order to tune the behavior. We'll also want to be able to measure the performance, to see how our tuning efforts pay off. This column will give you performance tuning guidelines, as well as sharpen your performance analysis and measurement skills.

The PGI Accelerator Fortran and C99 compilers available in PGI Release 9.0 target the NVIDIA CUDA-enabled GPUs and Tesla cards, and the tuning guidelines given here apply to those products. We'll use some of the enhanced features in the PGI 9.0-3 build, which should be available by the time you read this. We'll discuss tuning for other accelerator targets in future columns, as they are supported. The example programs here were run on our test system, a 2.67GHz,quad-core Intel Nehalem processor running Linux (RHEL 5.3), with an attached NVIDIA Tesla C1060 at 1.3GHz with 4GB memory. All the example programs are available from the PGI web site at www.pgroup.com/lit/samples/pginsider_v1n2a1_examples.tar.

Summary

Let's start with an executive summary. The most important part of a successful accelerated program is an appropriate parallel algorithm. NVIDIA GPUs are designed as throughput engines, to compute massive amounts of video data in parallel; they are designed to take advantage of the massive parallelism in several ways. First, GPUs can compute on different parts of the data set in parallel on different cores. Second, GPUs are designed to be more efficient by executing threads in groups, which NVIDIA calls *warps*; you can loosely think of a warp as a group of threads that execute in SIMD or vector mode, though that's not really how it's implemented. Third, the GPU can use the extra parallelism to keep the cores busy even when threads are waiting on some long latency operation, such as memory. With enough parallelism, GPU cores don't stall, unlike a CPU waiting for a cache miss.

In a PGI Accelerator Model program, parallelism is expressed in parallel loops. A high performance program will have one or more parallel loops, with lots (and lots) of iterations. If there is a single parallel loop with thousands of iterations, it can be successfully and effectively converted to an efficient GPU kernel. On the other hand, a single parallel loop with only a hundred or so iterations does not contain enough parallelism to keep the accelerator busy; nested parallel loops are necessary to generate enough parallelism with small loop limits.

Once you have an appropriate algorithm, you may want to tune the data movement between the host and the accelerator. The PGI compilers will automatically determine which arrays or array sections need to be sent to the accelerator from the host, and which modified arrays or array sections need to come back. However, a savvy programmer can tune the compiler analysis in two ways. First, the compiler tries to minimize the data traffic by sending the smallest array sections in either direction; this will often send a noncontiguous section of an array, such as the interior of a rectangular matrix. Because of the way the data transfers are done, it's usually much more efficient to send one large contiguous section; even if it means sending more data bytes, one transfer may map to a single DMA operation and be faster overall. A programmer can direct the compiler to do this with a data clause on the accelerator region directive. Second, the compiler (currently) will always bring data that is modified on the GPU back to the host. If that data isn't used after the accelerator region, the programmer can direct the compiler not to bring the data back, again with a data clause on the region directive. We'll see examples of these along with their effectiveness.

Then we must look at the performance of the parallel loops, or the kernel(s). First, look at the messages about non-stride-1 array accesses. NVIDIA GPUs access the device memory as 64-byte memory lines or superwords. The compiler tries to optimize the loop schedule to maximize the number of stride-1 array accesses, and it gives a message when there are one or more non-stride-1 accesses. You may be able to force stride-1 accesses by modifying the loop schedule, or you may want to change the array layout, transposing array dimensions, to get better memory performance

can't do such a search, since it can't run the program. The PGI compilers have an optimization procedure, which we're constantly tuning, but there are many cases where a programmer can get better performance by manually tuning the schedule.

Your GPU

First, you should know what kind of NVIDIA GPU you are targeting. NVIDIA has a broad family of CUDA-enabled GPUs, which differ in many respects. The high end cards, like our Tesla, have 30 multiprocessors and 4GB device memory; they have a *compute capability* of 1.3, which means they have hardware to support double-precision operations and better memory coalescing. Lower end cards will have fewer multiprocessors, less device memory, and a lower compute capability. You can see the features of your card using the `pgaccelinfo` command that comes with the PGI compilers. The output will look something like:

```
Device Number:                0
Device Name:                  Tesla C1060
Device Revision Number:       1.3
Global Memory Size:           4294705152
Number of Multiprocessors:    30
Number of Cores:              240
  .
  .
  .
```

The Device Revision Number is the compute capability, telling us that this card has double-precision support. If you have a revision 1.2 or lower, you can only run single-precision floating point on the card. There are also improvements in the way the GPU accesses memory with higher revisions, as we'll explain later. You can expect that a card with more multiprocessors will run your compute kernels that much faster than a card with fewer.

Appropriate Algorithm

To successfully run your program on a GPU, you must have an appropriately parallel algorithm. It must have lots of data parallelism, meaning that the program does the same operation over lots of data. In your program, this means a parallel loop, or even better, nested parallel loops. Let's take for example a four-point difference equation, here written in C:

```
for( i = 1; i < n-1; ++i )
   for( j = 1; j < m-1; ++j )
      a[i][j] = w0 * a[i][j] +
                w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

This nested loop has no parallelism; the `j` loop *carries* a dependence because of the assignment to `a[i][j]` in one `j` iteration, and the use of that value in the next `j` iteration when fetching `a[i][j-1]`. Similarly, the `i` loop carries a dependence because of the assignment to `a[i][j]` in one `i` iteration, and the use of that value when fetching `a[i-1][j]` for the next `i` iteration. If we put this loop in an accelerator region and build with `pgcc -ta=nvidia -Minfo`, we'll see messages to that effect:

```
test:
     32, No parallel kernels found, accelerator region ignored
     34, Loop carried dependence of 'a' prevents parallelization
         Loop carried backward dependence of 'a' prevents vectorization
     35, Loop carried dependence of 'a' prevents parallelization
         Loop carried backward dependence of 'a' prevents vectorization
```

At line 35 (the `j` loop), the messages tell us there are loop carried dependences for the array `a` in the inner loop; there are two messages because the dependence described above from the left hand side to the right hand side expression is treated as a lexically backward dependence, since the left hand side is assigned after the right hand side has been evaluated. There is also a loop carried write-after-read dependence from the use of `a[i][j+1]` to its reassignment in the next `j` iteration. Similarly, line 34 (the `i` loop) has loop carried dependences as well. Since there are no parallel loops, the compiler fails to generate any kernels, and ignores the accelerator region entirely.

However, we may be able to recast this as a parallel loop by using a Jacobi method, instead of the Gauss-Seidel method:

```
for( i = 1; i < n-1; ++i )
```

```
    for( j = 1; j < m-1; ++j )
      b[i][j] = w0*a[i][j] +
                w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      a[i][j] = b[i][j];
```

In this formulation, both loops around the first (`b`) assignment are parallel, because no values being assigned are used in the right hand side. Similarly, both loops around the second (`a`) assignment are parallel, for the same reason.

These difference equations are used in iterative computations; the Jacobi method is known to require more iterations to converge, but you may be able to improve your total performance even so, by computing each iteration in parallel. To generalize, taking the best sequential program and simply parallelizing it may not produce the best parallel program. You need to balance any overhead of a parallel program against the speedup you get by running in parallel. On today's GPUs, there's a lot of parallelism available, so it can absorb a lot of overhead.

The loops that will run well on a GPU have many parallel iterations. Nested loops will have two or more levels of parallelism. Note that the inner loop limits must be invariant in the outer loops; the GPU structures for nested parallelism require a rectangular domain. For efficient memory access, the arrays should have stride-1 access in one of the loops; the compiler tries to optimize for stride-1 accesses by reordering the nested loops, but it won't rearrange the data in memory. The body of the loop can be as small as a single statement, or it can be quite large, including nested internal loops; more on this later. Today's GPUs don't support procedure calls, so any functions called in the loop body will have to be inlined, manually by the programmer or automatically by the compiler.

Hindrances to GPU Parallelism

As you experiment with and port programs to the GPU, you will run into many cases where the compiler fails to generate parallel code. The compiler will produce informational messages (with `-Minfo`) that you'll want to know how to read. Here, we'll explain the most common messages you'll run into, and how to work around them.

```
 48, Loop carried scalar dependence for 'x'
```

This message can occur for several different reasons. One common idiom we use in programs is to save a value in a scalar for later reuse. For instance, instead of writing

```
for( i = 0; i < n-1; ++i ){
  b[i] = a[i] + a[i+1];
}
```

we might try to save one array fetch in the loop by using a scalar:

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

Such micro-optimization is particularly common in C programs, where it's even more likely that the programmer uses pointer arithmetic instead of array accesses. However, this second program has a loop-carried dependence for the scalar `x`. The value of `x` is assigned on one iteration and used on the next iteration, so the iterations are no longer independent. The original loop was completely parallel, so the problem is not the algorithm, it's just the program.

Another example comes from conditional assignments to a scalar.

```
for( i = 0; i < n; ++i ){
  if( a[i] < 0 ) x = b[i]*w;
  c[i] = x*a[i];
}
```

In this example, the scalar `x` is only assigned on some of the iterations; for those iterations where it isn't assigned, the value must come from some previous iteration, again hindering parallelism. This can even occur when the use is also conditional:

```
for( i = 0; i < n; ++i ){
  if( a[i] < 0 ) x = b[i]*w;
  d[i] = a[i]+1;
  if( a[i] < 0 ) c[i] = x*a[i];
}
```

In this example, the assignment and use are both conditional. The compiler, however, isn't clever enough to determine that the use only occurs under the same condition as the assignment, and so the value from the previous iteration is never needed. Rewriting this so the assignment and use are under the same condition removes the problem:

```
for( i = 0; i < n; ++i ){
  if( a[i] < 0 ){
    x = b[i]*w;
    c[i] = x*a[i];
  }
  d[i] = a[i]+1;
}
```

The current release of the compiler will give the same message for reduction loops as well, such as:

```
sum = 0.0;
for( i = 0; i < n; ++i ){
  sum += a[i] * b[i];
}
```

There are well-known methods to parallelize reductions, and future releases of the PGI Accelerator compilers will implement these automatically.

```
 Scalar last value needed after loop for 'x'
```

This message occurs when a scalar assigned in the loop is *live* after the loop; that is, its value is used (or the compiler thinks it might be used) after the loop.

```
for( i = 0; i < n; ++i ){
  x = b[i]*w;
  if( x != 0 ) c[i] = x*a[i];
}
if( x > 0 ){...
```

The loop itself may be parallel, but to do so the compiler has to *privatize* the scalars (make a private copy for each iteration), and the values are lost after the loop. Usually in these cases, the last value isn't really needed; perhaps the variable is a global variable (C `extern`), or there's some other use of the variable where the compiler can't prove that the value from the loop is never used again. Renaming the variable inside the loop so it's distinct from other variables in the program will solve this problem, or inserting a reassignment after the loop so the compiler knows that the value from the loop never reaches any uses outside is just as good.

```
 Loop carried dependence of 'a' prevents parallelization
```

This message comes out when there is a dependence due to an array reference. For C programs, we'll assume that all array pointers are declared with the `restrict` modifier, or the program is compiled with the `-Msafeptr` option; otherwise, the compiler can't tell whether any two array pointers might conflict. It might be a recurrence relation, or a simpler dependence from one iteration to another, or there might be a complex array subscript or an array that isn't indexed by the loop index. A real recurrence relation is shown below, where the value assigned in one iteration is used to compute the next value:

```
for( i = 1; i < n; ++i )
  a[i] = a[i-1]*b[i] + c[i];
```

In this case, there is a real dependence; there are some very complex methods to directly parallelize loops like this, but they are relatively inefficient and can suffer from numerical accuracy problems. There are no easy solutions for these cases.

A simpler case is the following:

```
for( i = 1; i < n; ++i ){
  a[i] = b[i] + c[i];
  d[i] = a[i-1] + 2*a[i];
}
```

Here, there is a loop-carried dependence because of the assignment to `a[i]` and its use in the next iteration as `a[i-1]`. One way to work around the problem in this case is to split the loop into two loops:

```
for( i = 1; i < n; ++i )
  a[i] = b[i] + c[i];
for( i = 1; i < n; ++i )
  d[i] = a[i-1] + 2*a[i];
```

Each of the two loops is now parallel; there is some cost since the `d` assignment has to fetch both operands from memory, but parallelism will often trump the small inefficiency.

When index arrays are used, the compiler has to assume there might be conflicts between iterations:

```
for( i = 0; i < n; ++i )
  a[ndx[i]] += b[i];
```

Sometimes, the operation can be recast so the index array is used on the right hand side:

```
for( i = 0; i < n; ++i )
  a[i] += b[rndx[i]];
```

Since the array `b` isn't assigned in the loop, there are no dependence conflicts.

Finally, we might have a case with a one-dimensional array in a doubly nested loop:

```
for( i = 0; i < n; ++i ){
  a[i] = 0.0;
  for( j = 0; j < n; ++i ){
    if( b[i][j] > 0.0 ) a[i] += b[i][j];
  }
}
```

In this case, the conditional assignment to `a` in the `j` loop can't be done in parallel; the left hand side doesn't use the `j` index at all, so each iteration is trying to assign the same element. However, the outer `i` loop can be successfully parallelized.

```
 Parallelization would require privatization of array 'a[0:n-1]'
```

This message is produced when the compiler notices that the array is a candidate for adding to a `private` clause for the loop. It's safe to do this if all uses of that array come from assignments in that loop, and the array isn't used after the loop. As noted above, the compiler automatically privatizes scalars, but currently doesn't automatically privatize arrays. An example is:

```
for( i = 0; i < n; ++i ){
  for( j = 0; j < n; ++i ){
    a[j] = b[i][j] * w + c[i]*d[j];
    if( a[j] > 0 ) p[i] -= a[j];
  }
```

```
}
```

Here, the uses of `a[j]` come from the assignment in the loop. The `j` loop can't be parallelized because of the conditional assignment to `p[i]`, which doesn't use the `j` index. However, if the array `a` isn't used after the loop, it can be privatized, allowing the compiler to parallelize the outer `i` loop. We can do this by adding the loop directive:

```
#pragma acc for private(a[0:n-1])
```

In the accelerator model, the compiler converts the body of some loop (or loops) into a kernel or kernels. Sometimes, the way the program is written prevents the obvious mapping. For instance, a conditional that branches around an inner loop will prevent nested loops from both executing in parallel.

```
for( i = 0; i < n; ++i )
   if( c[i] > 0 )
      for( j = 0; j < n; ++j )
         a[i][j] += b[i][j] * c[i];
```

Here, the compiler can either turn the outer `i` loop body into a kernel, running the `i` loop in parallel, or turn the inner `j` loop body into a kernel, running the `j` loop in parallel. However, it can't run both loops in parallel because the inner loop is only conditionally executed. Rewriting this as:

```
for( i = 0; i < n; ++i )
   for( j = 0; j < n; ++j )
      if( c[i] > 0 )
         a[i][j] += b[i][j] * c[i];
```

Allows both loops to execute in parallel.

Accelerator and Compiler Restrictions

There are a number of restrictions on what is allowed in an accelerated loop for the NVIDIA GPUs; some of these are limitations of the device, some are limitations imposed by the compiler. You might see messages such as:

```
Accelerator restriction: unsupported operation
Accelerator restriction: function/procedure calls are not supported
Accelerator restriction: struct/member references are not yet supported
Accelerator restriction: datatype not supported
```

Some of these restrictions will be lifted in future compiler releases; until then, that part of the program will have to be left on the host, or rewritten to avoid the restriction.

```
Accelerator compiler license not found, accelerator code generation disabled.
```

If you see this message, then it's likely that you either haven't received or haven't installed the license for the accelerator features, or that perhaps it's expired. Either install your license, or contact PGI Support for an updated license.

Parallelizable Loops

```
Loop is parallelizable
```

The compiler notes which loops are potentially parallelizable with this message. This doesn't mean the loop will be executed in parallel, just that there are no dependences that would prevent it. We will show below why the compiler might choose not to run a parallelizable loop in parallel.

```
Accelerator kernel generated
```

This message appears with the line number of the top of some loop; it tells you that the body of that loop will be the kernel, and that the enclosing loops will be executed in parallel according to the loop schedule described just after this

message. We'll describe the loop schedules below.

Performance Analysis

Once you have an appropriately parallel program that successfully generates accelerator kernels, you can start to look at performance. Here, we'll look at three ways to inspect the performance. We'll look at the compiler messages below, when we start looking at specific performance issues. We'll run the program with the profile library. And finally, we'll run the program under control of NVIDIA's `cudaprof` tool.

PGI Release 9.0 includes a simple accelerator profile feature, enabled by adding an option to the target accelerator command line flag. By linking with the `-ta=nvidia,time` flag, the profile library is linked into the program. When the program is run, performance is collected for accelerator regions and kernels. Specifically, it collects the elapsed time spent in the accelerator region and in each kernel in the region, and separates out compute time from host-GPU communication time. It also shows how much time is spent initializing the device; we'll show a little trick to reduce that as well.

After building with `-ta=nvidia,time` and running the program, the profile library will print out a performance summary that looks something like the following:

```
Accelerator Kernel Timing data
c5.c
  test
    32: region entered 1 time
        time(us): total=1411909 init=1408006 region=3903
                  kernels=44 data=3859
        w/o init: total=3903 max=3903 min=3903 avg=3903
        35: kernel launched 1 times
            grid: [7x7]  block: [16x16]
            time(us): total=28 max=28 min=28 avg=28
        39: kernel launched 1 times
            grid: [7x7]  block: [16x16]
            time(us): total=16 max=16 min=16 avg=16
```

The information here tells me that the program only executed one accelerator region, at line 32 in routine `test` in file `c5.c`. It entered that region only once and spent 1.412 seconds in the region. Of that, 1.40 seconds were spent initializing the device, and 4ms spent executing the kernels in the region. Most of that time was spent moving data, only 44 microseconds spent actually executing kernel code. The region had two kernels, at lines 35 and 39, each executed once.

Initialization Time

To optimize the performance of this code, it's clear we want to first address the initialization cost, then try to reduce the time spent moving data. The initialization time here is the time spent in the NVIDIA runtime library and device drivers to initialize memory structures and connect to the device. Normally, this cost only needs to be paid once per program execution. In this case, the initialization is over a second; we've seen it range from 40 milliseconds up to 8 seconds on one cluster installation. For large jobs, this overhead isn't really important, but we'd like to reduce it if it's easy. We're not quite sure why it's so high in some cases and low in others, but we've found that if you have two jobs running in parallel, the initialization for the second job is much less expensive. Apparently most of the time is spent initializing static structures in the device driver, which are kept active as long as there's a process connected to the driver. PGI release 9.0-3 includes a utility program `pgcudainit`; if you run this program in background mode, it will hold open a CUDA connection to the device driver, significantly reducing initialization time for subsequent programs. For our program, run with `pgcudainit` activated in background mode, the timing is:

```
Accelerator Kernel Timing data
c5.c
  test
    32: region entered 1 time
        time(us): total=91628 init=88817 region=2811
                  kernels=44 data=2767
        w/o init: total=2811 max=2811 min=2811 avg=2811
        35: kernel launched 1 times
            grid: [7x7]  block: [16x16]
            time(us): total=28 max=28 min=28 avg=28
        39: kernel launched 1 times
```

```
        grid: [7x7]  block: [16x16]
        time(us): total=16 max=16 min=16 avg=16
```

The initialization time was reduced from 1.4 second to less than 1/10 second.

Host / Accelerator Data Movement

Now we address the data communication between the host and accelerator. In our example above, data transfer took about 60X as much time as the computation. We should first look at the data transfer `–Minfo` messages from the compiler; for this program we see:

```
   32, Generating copyout(b[1:n-2][1:m-2])
       Generating copyin(a[0:n-1][0:m-1])
       Generating copyout(a[1:n-2][1:m-2])
```

This example program is the Jacobi iteration example from above. We note two things here. First, the array `b` is only used as a temporary; its values are not needed back on the host after the loop. As mentioned earlier, the compiler will, by default, always bring modified values back to the host. You can override this by adding an appropriate clause to the accelerator region directive:

```
#pragma acc region local(b[1:n-2][1:m-2])
```

This tells the compiler to allocate the space for the array `b`, but not to generate any data copies for it. When we build the modified program, the compiler message for the region is:

```
   32, Generating local(b[1:n-2][1:m-2])
       Generating copyin(a[0:n-1][0:m-1])
       Generating copyout(a[1:n-2][1:m-2])
```

When we run this version, the time spent for data traffic in our example program drops from about 2.7ms to 1.7ms.

The second thing we notice is the data traffic for the `a` array includes a noncontiguous region. The `copyin` generated is for the whole matrix, but the `copyout`, from the GPU back to the host, only moves the modified elements, which are the interior of the array. This minimizes the data traffic, but moving noncontiguous regions is more costly than moving one large contiguous section. We can tune this by adding another clause to the region directive:

```
#pragma acc region local(b[1:n-2][1:m-2]) copy(a[0:n-1][0:m-1])
```

This tells the compiler to move the whole `a` array both over to the GPU and back again; it moves more data, but the moves are more efficient. The messages from the compiler are now:

```
   32, Generating local(b[1:n-2][1:m-2])
       Generating copy(a[:n-1][:m-1])
```

and the time spent moving data drops from 1.7ms to .5ms. The total savings is 2.7ms to .5ms, about 5X improvement.

The improvement from copying whole arrays depends on the array `a` being allocated in a contiguous block. For C programs, the data layout for multidimensional arrays is under user control. To take advantage of contiguous data transfers, you have to allocate your array as a contiguous memory block. To be specific, suppose you allocate each column of your two-dimensional array separately, as shown here:

```
b = (float**) malloc( sizeof(float*) * n );
for( i = 0; i < n; ++i )
   b[i] = (float*) malloc( sizeof(float) * m );
```

Because `malloc` adds some control information to the memory blocks it manages, and because it aligns the blocks, two calls to `malloc` never return contiguous memory blocks. This means that your program will never be able to benefit from fast, contiguous DMA transfers between the host and the GPU. Instead, you can allocate the array as follows:

```
b = (float**) malloc( sizeof(float*) * n );
```

```
b[0] = (float*) malloc( sizeof(float) * m * n );
for( i = 1; i < n; ++i ) b[i] = b[i-1] + m;
```

Then the columns are adjacent, and the whole array is a single block of data. Fortran programs don't have this problem, because multidimensional arrays are an intrinsic part of the language.

So three rules for tuning Host-Accelerator data traffic are: First, build with the `-ta=nvidia,time` flag to see whether your program has a data traffic bottleneck. Second, look to see if any arrays are candidates to be local to the accelerator; those arrays would essentially be temporary arrays on the GPU, and are not needed back on the host. Use the `local` clause on the region directive for those arrays. Third, look to see if the compiler is sending noncontiguous subarrays in one direction or the other, and whether it's feasible to just transfer the whole array in one big chunk. If so, add the appropriate `copy`, `copyin`, or `copyout` clause to the region directive.

Kernel Performance

Then we start to look at the performance of the kernel itself. The simple profile information we get back from using the `-ta=nvidia,time` gives no details, so we have to look more deeply. We'll start by looking for compiler messages that give us clues as to the performance, particularly in regards to memory bandwidth utilization on the GPU itself.

An NVIDIA GPU has two kinds of memory. First is the large (up to 4GB, in today's cards) device memory, which holds all data allocated and initialized by the host. When a GPU kernel is running, data is fetched from the memory in what used to be called *superwords*; that is, 16 words (64 bytes) are fetched at a time. The threads on the GPU are executed in groups of 32 called *warps*; data is fetched for half a warp at a time (16 threads). For NVIDIA GPUs with compute capability 1.3, if all the data needed by a half-warp is contained in a single superword, then only that one superword is fetched. For stride-1 array accesses, the 16 threads will access 16 words that will take up one superword (if they are aligned) or two adjacent superwords (otherwise); NVIDIA calls this *memory coalescing*, when memory accesses from multiple threads are satisfied by one superword fetch. Non-stride-1 accesses will take more superword accesses; in the worst case, strides greater than 16, or indexed memory accesses, it can take up to 16 superword accesses. Since only one word of the 16 actually gets used, this wastes 15/16 (93%) of the available memory bandwidth. So it's quite important that the kernel be optimized for stride-1 accesses.

Cards with compute capability of 1.2 or lower have more stringent requirements. In such cards, memory coalescing only works when the data access is stride-1 and aligned on a 16-word boundary; that is, the 16 threads in a half-warp must access the 16 memory words in a single superword in the order that the words appear in memory.

So it's important to optimize for memory strides, and if possible, for memory alignment. The compiler will do this automatically, as much as it can. When it can't, it gives a message:

```
32, Non-stride-1 accesses for array 'a'
```

to flag non-stride-1 accesses; those are cases where you might want to inspect the program to see if you can reorganize the program or change the data layout to remove the non-stride-1 accesses.

The compiler also detects where it can use the software managed data cache to reduce the number of device memory accesses. You'll see a message that looks something like:

```
34, Cached references to size [16x16] block of 'a'
```

in those cases.

The current release does not optimize for data alignment. You can sometimes improve the overall performance of your program by padding the leading dimension of your array to a multiple of 16 elements, and using a data directive to send over that whole dimension, even though not all those elements are accessed.

Kernel Schedule

Our final step is to tune the kernel schedule. The kernel schedule tells how the parallel loops are mapped onto the hardware parallelism, and it's important to understand. For instance, if there is only a single parallel loop, the compiler will split (strip-mine) the loop into chunks or *strips* of some length. It will schedule each strip to execute in SIMD or *vector* mode on a single multiprocessor. It will then schedule the different strips to execute in *parallel* mode across the multiprocessors. In detail, the compiler will use `threadidx.x` to index the threads in a single vector strip, and `blockidx.x` to index the different strips; it will combine these to create the original loop index. In such a case, you'll see a compiler message like:

```
    Accelerator kernel generated
     33, #pragma for parallel, vector(256)
```

which tells you that the compiler generated vector strips of size 256, and the strips execute in parallel across the multiprocessors. It presents this information using the same directive syntax you would use if you wanted to force this schedule.

If you have two nested parallel loops, the compiler will inspect several loop schedules before choosing one; as mentioned, it tries to optimize memory accesses and total parallelism. Among the options it might choose are:

- Run the inner loop in *vector* mode and the outer loop in *parallel* mode:

```
Accelerator kernel generated
33, #pragma for parallel
34, #pragma for vector(256)
```

  The inner loop is actually strip-mined in this case as well, in case the trip count is greater than 256, with the strip loop executing sequentially on the GPU.
- Strip-mine the inner loop as before, running the outer loop in parallel mode:

```
Accelerator kernel generated
33, #pragma for parallel
34, #pragma for parallel, vector(256)
```

- Strip-mine both loops, running the strips of each loop in vector mode; note that the product of the vector strip sizes must not exceed 256:

```
Accelerator kernel generated
33, #pragma for parallel, vector(16)
34, #pragma for parallel, vector(16)
```

- Strip-mine both loops with different strip sizes:

```
Accelerator kernel generated
33, #pragma for parallel, vector(2)
34, #pragma for parallel, vector(64)
```

- Strip-mine the outer loop for parallel/vector execution, running the inner loop in parallel mode, perhaps to optimize for memory strides

```
Accelerator kernel generated
33, #pragma for parallel, vector(128)
34, #pragma for parallel
```

So why might you want to modify the schedule chosen by the compiler? You may know some things that the compiler does not. For instance, the compiler will generally try to generate long vector operations, to take advantage of as much parallelism as possible. Suppose it chooses to run a loop with vector strips of length 256, and you know that usually the trip count of that loop is less than 100; then the compiler is generating useless parallelism. In that case, it might be better to run that loop with a shorter vector length, and perhaps even enable vector parallelism along a second dimension, if available.

There are also performance nonlinearities with today's generation of GPUs. Small changes in the program can make measurable changes in performance. As mentioned above, the current state of the art for scheduling is to search through all the possible loop schedules and test the performance. Until we can train the compiler how to better predict the performance of a kernel, a savvy user will often be able to find a better schedule by trial and error.

Using cudaprof

NVIDIA provides a graphical `cudaprof` performance analysis tool. It runs your program several times and collects time information during the run. When you first fire it up, you'll want to start a new session (File:New), and give it a session name and a directory in which to store the cudaprof project file (suffix `.cpj`). You then give it the name of your executable file and working directory, any command line arguments, and a maximum time to allow it to execute. You can also select which counters to collect. When you start the program, the default is to run it five times and aggregate the data. The initial screen shows a trace of the calls to NVIDIA runtime routines and GPU kernels.

File Session View Options Window Help

Sessions

- Session1
- Session2
- Session3
  - Device_0
    - Context_0

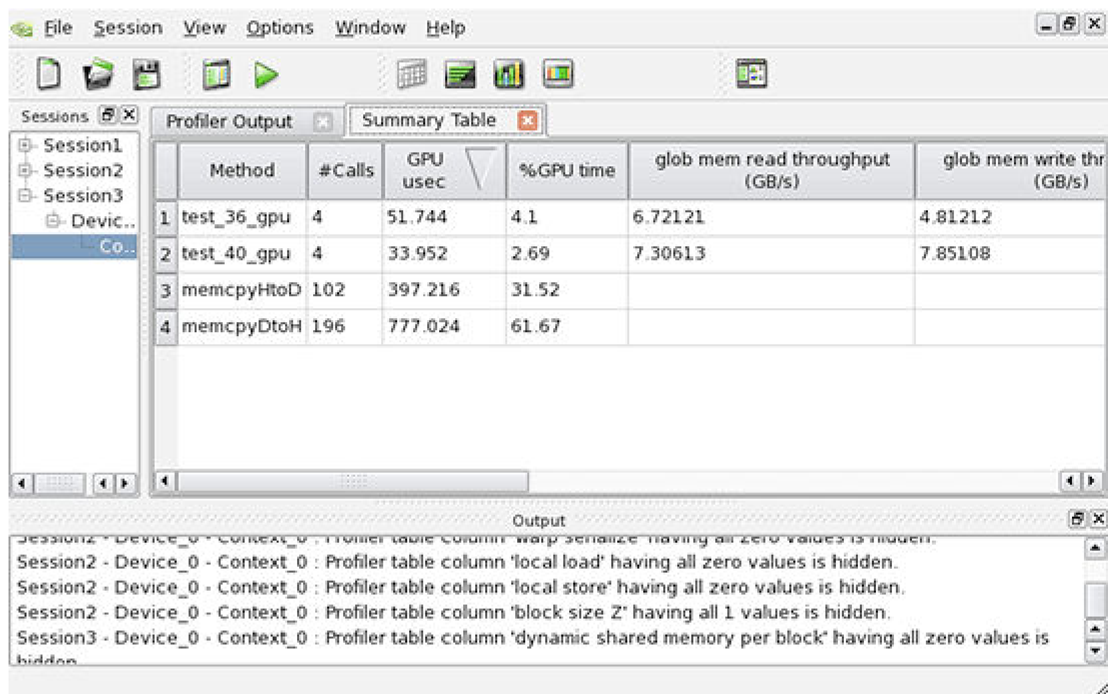
Profiler Output

	GPU Timestamp usec	Method	GPU Time	CPU Time	Occupancy	grid size
92	1282.56	memcpyHtoD	3.648			
93	1296.39	memcpyHtoD	4.096	1		
94	1310.73	memcpyHtoD	3.648			
95	1324.55	memcpyHtoD	4.096			
96	1338.62	memcpyHtoD	3.648			
97	1352.45	memcpyHtoD	3.648	1		
98	1366.53	memcpyHtoD	4.064	1		
99	1380.61	memcpyHtoD	3.68			
100	1394.44	memcpyHtoD	4.064	1		
101	1408.78	memcpyHtoD	3.776			
102	1422.86	memcpyHtoD	3.584			
103	1452.81	test_36_gpu	14.88	29	1	7
104	1520.14	test_40_gpu	8.448	19	1	7
105	1578.77	test_36_gpu	12.448	22	1	7
106	1641.22	test_40_gpu	8.576	18	1	7
107	1699.84	test_36_gpu	12.096	22	1	7
108	1761.28	test_40_gpu	8.512	18	1	7
109	1821.95	test_36_gpu	12.32	22	1	7
110	1887.5	test_40_gpu	8.416	18	1	7
111	1937.67	memcpyDtoH	4.672	15		
112	1953.28	memcpyDtoH	3.968	13		
113	1965.58	memcpyDtoH	3.968	10		
114	1979.91	memcpyDtoH	3.936	13		
115	1993.73	memcpyDtoH	3.904	13		
116	2008.33	memcpyDtoH	3.936	13		
117	2020.61	memcpyDtoH	3.936	10		

Output

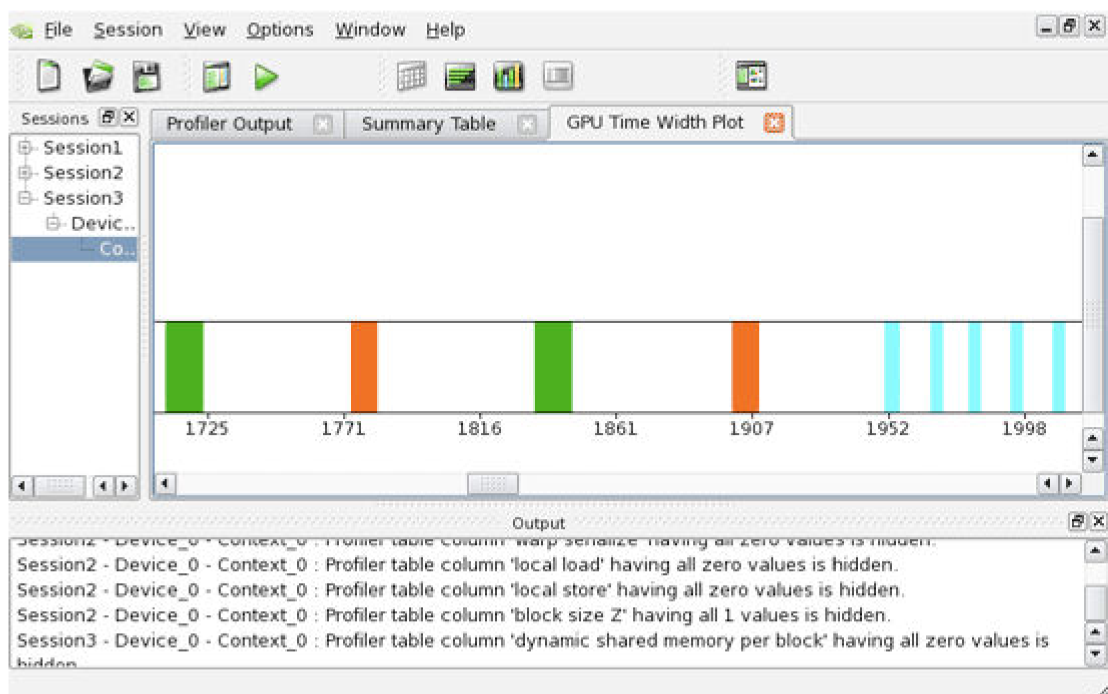
Session2 - Device_0 - Context_0 : Profiler table column 'local load' having all zero values is hidden.
Session2 - Device_0 - Context_0 : Profiler table column 'local store' having all zero values is hidden.
Session2 - Device_0 - Context_0 : Profiler table column 'block size Z' having all 1 values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'dynamic shared memory per block' having all zero values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'stream id' having all zero values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'warp serialize' having all zero values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'local load' having all zero values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'local store' having all zero values is hidden.
Session3 - Device_0 - Context_0 : Profiler table column 'block size Z' having all 1 values is hidden.

More interesting are the Summary Table, Width Plot, Height Plot and Summary Plot panes, accessible from the toolbar or under the View menu. The Summary table tells you how many times each kernel and each runtime routine was called, and how much time was spent aggregate in that routine; for the GPU kernels, it also can give a great deal of additional detailed information, such as branches, divergent branches, instruction throughput and more. Here, we see that the program, our Jacobi iteration kernel, is spending most of its time copying data between the host and GPU.

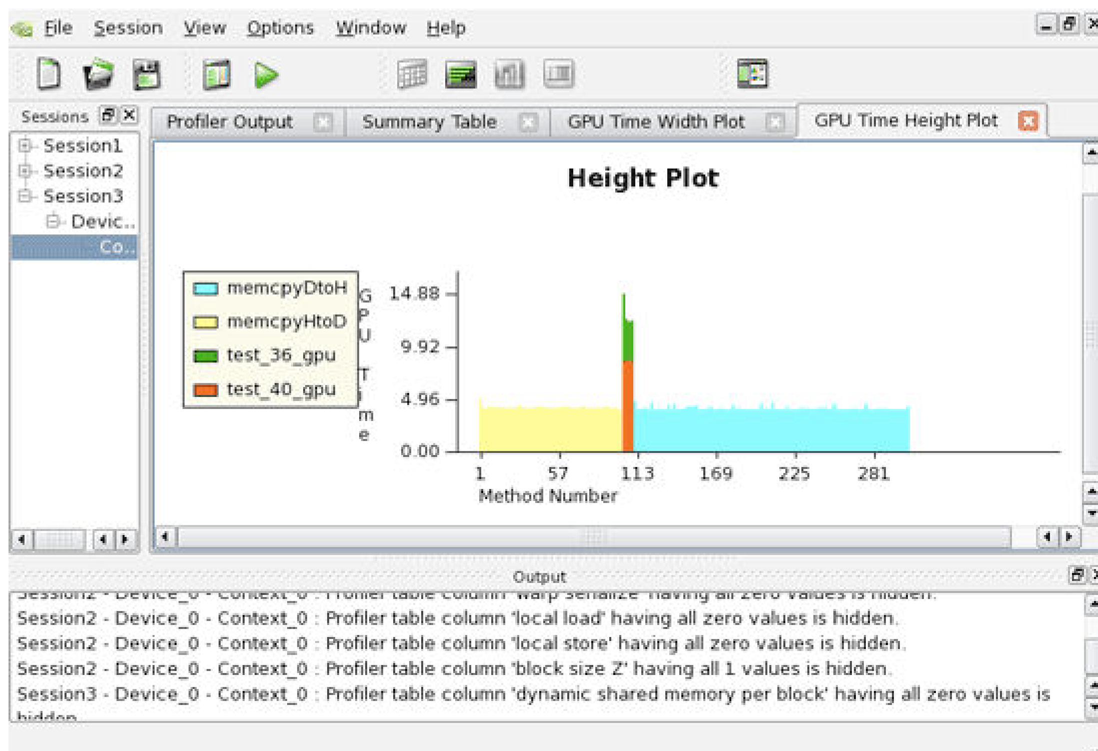


	Method	#Calls	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write thr (GB/s)
1	test_36_gpu	4	51.744	4.1	6.72121	4.81212
2	test_40_gpu	4	33.952	2.69	7.30613	7.85108
3	memcpyHtoD	102	397.216	31.52		
4	memcpyDtoH	196	777.024	61.67		

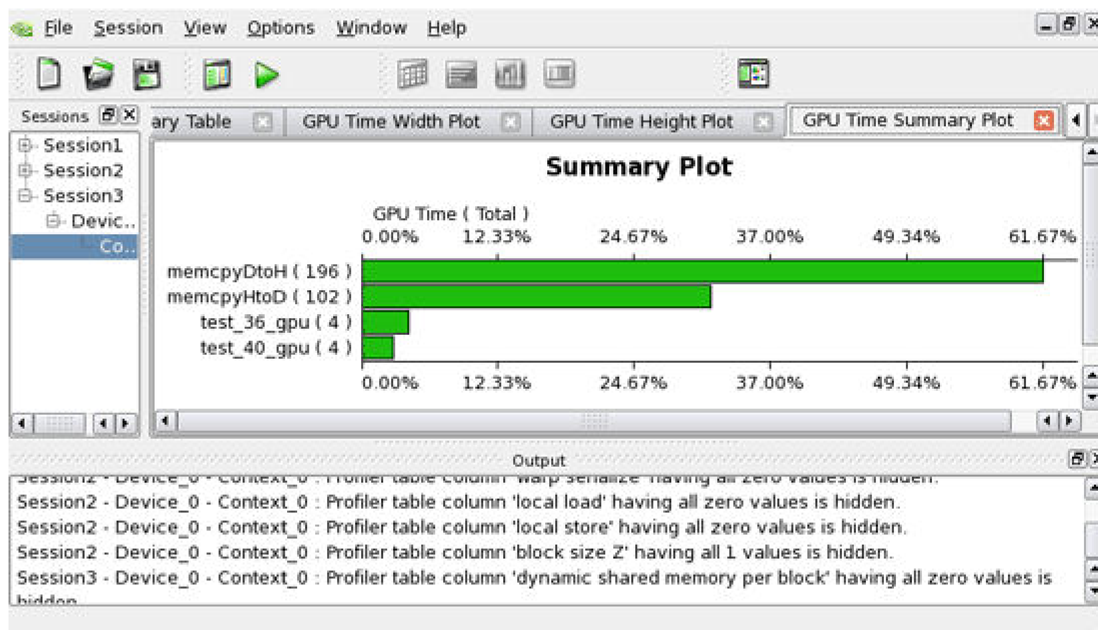
The Width Plot and Height Plot show the program trace in graphical form. The Width Plot shows time horizontally, here showing a couple of kernel calls and many calls to copy data back to the host, because the data is not contiguous.



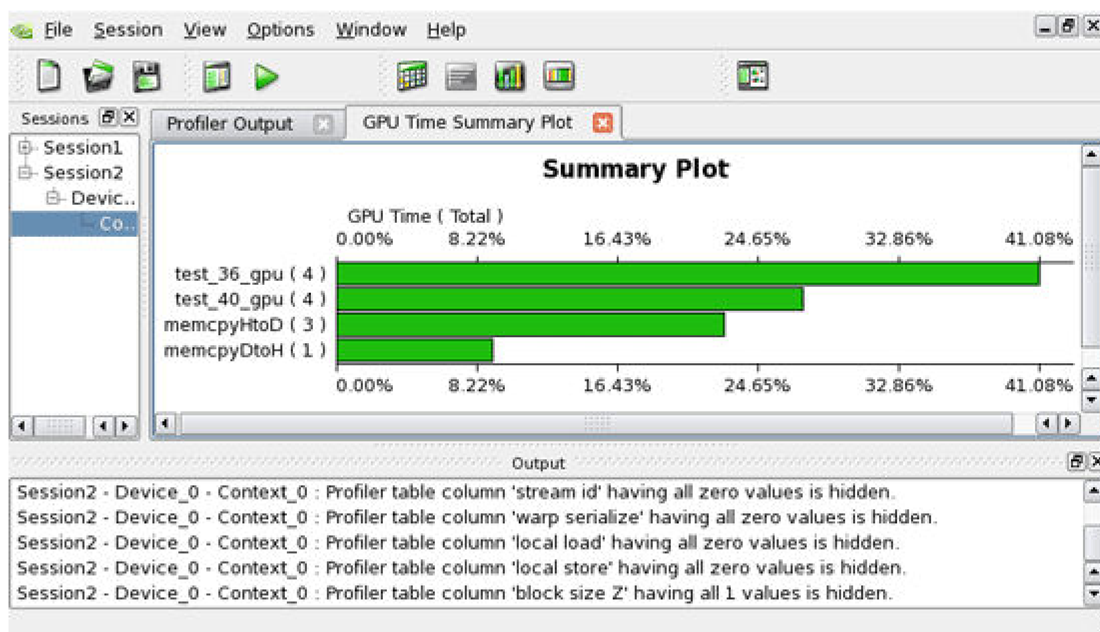
The Height Plot shows invocation time horizontally and time in that invocation vertically.



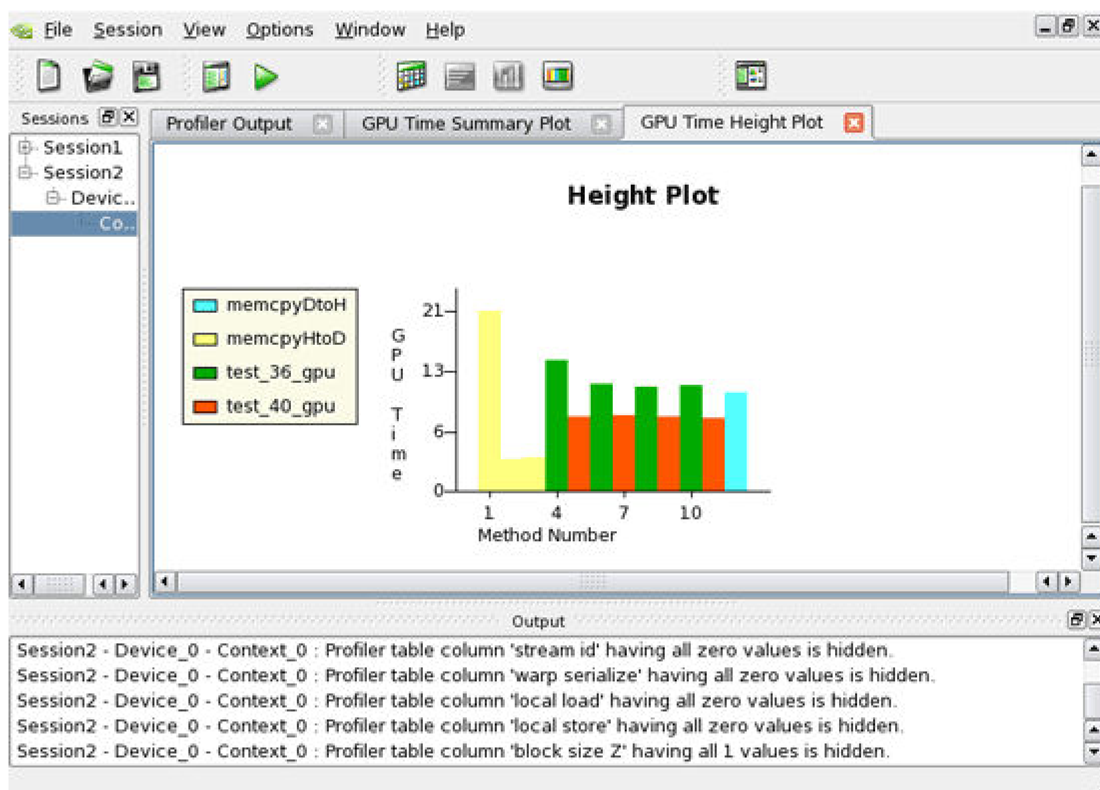
Finally, the Summary plot shows what fraction of time was spent in each routine or kernel.



We see the effect of adding the copy clauses to optimize the data movement, as described above, with the Summary Plot:



The Height plot shows that there are only three calls to copy data to the device, and one to copy data back, because all the data is now contiguous.



Summary

This installment looked at a four step process to tune performance of loops for NVIDIA GPUs: use an appropriate algorithm, tune the host/GPU data movement, optimize for strides and alignment, and understand and perhaps experiment with the kernel schedule. We hope you find the feedback from the compiler useful in your tuning process. We're continuing to work on both the compiler analysis and the feedback, as well as additional features to come in future releases. Our next installment will look in detail at our experiences porting a large application, where we have to address a number of interesting issues.