The Heterogeneous Programming Jungle

March 19, 2012 | Michael Wolfe, Compiler Engineer, The Portland Group, Inc.

There's a lot of talk now about heterogeneous computing, but here I want to focus on heterogeneous programming. There are several, perhaps many, approaches being developed to program heterogeneous systems, but I would argue that none of them have proven that they successfully address the real goal. This article will discuss a range of potentially interesting heterogeneous systems for HPC, why programming them is hard, and why developing a high level programming model is even harder.

I've always said that parallel programming is intrinsically hard and will remain so. Parallel programming is all about performance. If you're not interested in performance, save yourself the headache and just use a single thread. To get performance, your program has to create parallel activities, insert synchronization between them, and manage data locality.

The heterogeneous systems of interest to HPC use an attached coprocessor or accelerator that is optimized for certain types of computation. These devices typically exhibit internal parallelism, and execute asynchronously and concurrently with the host processor. Programming a heterogeneous system is then even more complex than "traditional" parallel programming (if any parallel programming can be called traditional), because in addition to the complexity of parallel programming on the attached device, the program must manage the concurrent activities between the host and device, and manage data locality between the host and device.

A Heterogeneous System Zoo

Before we embark on a discussion of the jungle of programming options, it's instructive to explore the range of heterogeneous systems that are currently in use, being designed, or being considered, from GPUs to Intel MIC, DSPs and beyond.

- Intel/AMD X86 host + NVIDIA GPUs (x86+GPU). This is the most common heterogeneous HPC system we see today, including 35 of the Top 500 supercomputers in the November 2011 list. The GPU has its own device memory, which is connected to the host via PCIe. Data must be allocated on and moved to the GPU memory, and parallel kernels are launched asynchronously on the GPU by the host.
- AMD Opteron + AMD GPUs. This is essentially another x86+GPU option, using AMD Firestream instead of NVIDIA. The general structure is the same as above.
- AMD Opteron + AMD APU. The AMD Heterogeneous System Architecture (formerly AMD Fusion) could easily become a player in this market, given that the APU (Accelerated Processor Unit) is integrated on the chip with the Opteron cores. Current offerings in this line are programmed much like x86+GPU. The host and APU share physical memory, but the memory is partitioned. It's very like the x86+GPU case, except that a data copy between the two memory spaces can run at memory speeds, instead of at PCI speed. Again, parallel kernels are launched asynchronously on the APU by the host. AMD has announced aggressive plans for this product line in the future.
- Intel Core + Intel Ivy Bridge integrated GPU. The on-chip GPU on the next generation Ivy Bridge processor is reported to be OpenCL programmable, allowing heterogeneous programming models as well. Intel's Sandy Bridge has an integrated GPU, but it seems not to support OpenCL.
- Intel Core + Intel MIC. Reportedly, the MIC is *the* highly parallel technical computing accelerator architecture for Intel. Currently, the MIC is on a PCIe card like a GPU, so it has the same data model as x86+GPU. The MIC could support parallel kernels like a GPU, but it also can support OpenMP parallelism or dynamic task parallelism among the cores on the chip as well. An early Intel Larrabee article (also published in PLDI 2009) described a programming model that supported virtual shared memory between

the host and Larrabee, but I haven't heard whether the MIC product includes support for that model.
- NVIDIA Denver: ARM + NVIDIA GPU. As far as I know, this is not yet a product, and could look like the AMD APU, except for the host CPU and accelerator instruction sets.
- Texas Instruments: ARM + TI DSPs. A recent HPCwire article described TI's potential move (return, actually) to HPC using multiple 10GHz DSP chips. It's pretty early to talk about system architecture or programming strategy for these.
- Convey Intel x86 + FPGA-implemented reconfigurable vector unit. I've always thought of the Convey machine as a coprocessor or accelerator system with a really interesting implementation. One of the key advantages of the system is that the coprocessor memory controllers live in the same virtual address space as the host memory. It's separate physical memory, but the host can access the coprocessor memory, and vice versa, though with a performance cost. Moreover, the programming model looks more like vector programming than like coprocessor offloading.
- GP core + some other multicore or stream accelerators. Possibilities include a Tilera multicore or the Chinese FeiTeng FT64. Again, it's a little early to talk about system architecture or programming strategy.
- GP core + FPGA fabric. Convey uses FPGAs to implement the reconfigurable vector unit, but you could consider a more customizable unit as well. There were plenty of FPGA products being displayed at SC11 in Seattle, but none were as well integrated as Convey's, nor did they have as strong a programming environment. The potential for fully custom functional units with application-specific, variable-length datatypes is attractive in the abstract, but except for a few success stories in financial and bioinformatics, this approach has a long way to go to gain much traction in HPC.
- IBM Power + Cell. This was more common a couple years ago. While there are still a few Cell-equipped supercomputers on the Top 500 list, IBM has pulled the plug on the PowerXCell 8i product, so expect these to vanish.

In the HPC space, there seem to be only two lonely vendors still dedicated to CPU-only solutions: IBM and the Blue Gene family, and future Fujitsu SPARC-based systems, follow-ons to the Fujitsu K computer.

Similarities

Despite the wide variety of heterogeneous systems, there is surprising similarity among most or all of the various designs. All the systems allow the attached device to execute asynchronously with the host. You would expect this, especially since most of the devices are programmable devices themselves, but even the tightly-connected Convey coprocessor unit executes asynchronously.

All the systems exhibit several levels of parallelism within the coprocessor.

- Typically, the coprocessor has several, and sometimes many, execution units (I'll avoid over-using the word *processor*). NVIDIA Fermi GPUs have 16 Streaming Multiprocessors (SMPs); AMD GPUs have 20 or more SIMD units; Intel MIC will have 50+ x86 cores. If your program doesn't take advantage of at least that much parallelism, you won't be getting anywhere near the benefit of the coprocessor.
- Each execution unit typically has SIMD or vector execution. NVIDIA GPUs execute threads in SIMD-like groups of 32 (what NVIDIA calls *warps*); AMD GPUs execute in *wavefronts* that are 64-threads wide; the Intel MIC, like its predecessor Larrabee, has 512-bit wide SIMD instructions (16 floats or 8 doubles). Again, if your application doesn't take advantage of that much SIMD parallelism, your performance is going to suffer by the same factor.

Since these devices are used for processing large blocks of data, memory latency is a problem. Cache memory doesn't help when the dataset is larger than the cache. One method to address the problem is to use a high bandwidth memory, such as Convey's Scatter-Gather memory. The other is to add multithreading, where the execution unit saves the state of two or more threads, and can swap execution between threads in a single cycle. This strategy can range from swapping between threads at a cache miss, or alternating between threadson every cycle. While one thread is waiting for memory, the execution unit keeps busy by switching to a different thread. To be effective, the program needs to expose even more parallelism, which will be exploited via multithreading. Intel Larrabee / Knights Ferry has four-way multithreading per core, whereas GPUs have a multithreading factor of 20 or 30. Your program will need to provide a factor of somewhere between four and twenty times more parallel activities for the best performance. Moreover, the programming model may want to

expose the difference between multithreading and multiprocessing. When tuning for a multithreaded multiprocessor, the programmer may be able to share data among threads that share the same execution unit, since they will share cache resources and can synchronize efficiently, and distribute data across threads that use different execution units.

But most importantly, the attached device has its own path to memory, usually to a separate memory unit. These designs fall into three categories:

- Separate physical memory: Current discrete GPUs and the upcoming Intel MIC have a separate physical memory connected to the attached device, not directly connected to the host processor, and often even not accessible from the host. The device is implemented as a separate card with its own memory. There may be support for the device accessing host memory directly, or the host accessing device memory, but at a significant performance cost.
- Partitioned physical memory: Today's AMD Fusion processor chips fall into this category, as do low-end systems (such as laptops) using a motherboard-integrated GPU. There is one physical memory, but some fraction of that memory is dedicated to the APU or GPU. Logically, it looks like a separate physical memory, except copying data between the two spaces is faster, because both spaces are on the same memory bus, and moving data to the APU or GPU is slower, because the CPU main memory doesn't have the bandwidth of, say, a good graphics memory system typical for a GPU.
- Separate physical memory, one virtual memory: The Convey system fits this category. The coprocessor has its own memory controllers to its own memory subsystem, but both the CPU and coprocessor physical memories are mapped to a single virtual address space. It becomes part of application optimization to make sure to allocate data in the coprocessor physical memory, for instance, if it will be mostly accessed by coprocessor instructions.

Programming Model Goals

Given the similarities among system designs, one might think it should be obvious how to come up with a programming strategy that would preserve portability and performance across all these devices. What we want is a method that allows the application writer to write a program once, and let the compiler or runtime optimize for each target. Is that too much to ask?

Let me reflect momentarily on the two gold standards in this arena. The first is high level programming languages in general. After 50 years of programming using Algol, Pascal, Fortran, C, C++, Java, and many, many other languages, we tend to forget how wonderful and important it is that we can write a single program, compile it, run it, and get the same results on any number of different processors and operating systems.

Second, let's look back at vector computing. The HPC space 30 years ago was dominated by vector computers: Cray, NEC, Fujitsu, IBM, Convex, and more. Building on the compiler work from very early supercomputers (such as TI's ASC) and some very good academic research (at Illinois, Rice, and others), these vendors produced vectorizing compilers that could generate pretty good vector code from loops in your program. It was not the only way to get vector performance from your code. You could always use a vector library or add intrinsics to your program. But vectorizing compilers were the dominant vector programming method.

Vectorizing compilers were successful for three very important reasons. The compilers not only attempted to vectorize your loops, they gave some very specific user feedback when they failed. How often does your optimizing compiler tell you when it failed to optimize your code? Never, most likely. In fact, you would find it annoying if it printed a message every time it couldn't float a computation out of a loop, say. However, the difference between vector performance and non-vector performance was a factor of 5 or 10, or more in some cases. That performance should not be left on the table. If a programmer is depending on the compiler to generate vector code, he or she really wants to know if the compiler was successful. And the feedback could be quite specific. Not just *failed to vectorize the loop at line 25*, but *an unknown variable N in the second subscript of the array reference to A on line 27 prevents vectorization*. So the first reason vectorizing compilers were successful is that the programmer used this feedback to rewrite that portion of the code and eventually reach vector performance. The second reason is that this feedback slowly trained the programmer how to write his or her next program so it would vectorize automatically.

The third reason, and ultimately the most important, is that the style of programming that vectorizing compilers promoted gave good performance across a wide range of machines. Programmers learned to make the inner loops be stride-1, avoid conditionals, inline functions (or sink loops into the functions), pull I/O out of the inner loops, and identify data dependences that prevent vectorization. Programs written in this style would then vectorize with the Cray compiler, as well as the IBM, NEC, Fujitsu, Convex, and others. Without ever collaborating on a specification, the vendors trained their users on how to write performance-portable vector programs.

I claim that what we want is a programming strategy, model or language that will promote programming in a style that will give good performance across a wide range of heterogeneous systems. It doesn't necessarily have to be a new language. As with the vectorizing compilers lesson, if we can create a set of coding rules that will allow compilers and tools to exploit the parallelism effectively, that's probably good enough. But there are several factors that make this hard.

Why It's Hard

Parallel programming is hard enough to begin with. Now we have to deal not only with the parallelism we're accustomed to on our multicore CPUs, we have to deal with an attached asynchronous device, as well as with the parallelism on that device.

To get high performance parallel code, you have to optimize locality and synchronization as well. For these devices, locality optimization mostly boils down to managing the distinct host and device memory spaces. Someone has to decide what data gets allocated in which memory, and whether or when to move that data to the other memory.

Many systems will have multiple coprocessors at each node, each with its own memory. Users are going to want to exploit all the resources available, and that may mean managing not just one coprocessor, but two or more. Suddenly you have not just a data movement problem, but a data distribution problem, and perhaps load balancing issues, too. These are issues that were addressed partly by High Performance Fortran in the 1990s. Some data has to be distributed among the memories, some has to be replicated, and some has to be shared or partially shared. And remember that these coprocessors are typically connected to some pretty high performance CPUs. Let's not just leave the CPU idle while the coprocessor is busy, let's distribute the work (and data) across the CPU cores as well as the coprocessor(s).

Most of the complexity comes from the heterogeneity itself. The coprocessor has a different instruction set, different performance characteristics, is optimized for different algorithms and applications than the host, and is optimized to work from its own memory. The goal of HPC is the **HP** part, the high performance part, and we need to be able to take advantage of the features of the coprocessor to get this performance.

The challenge of designing a higher level programming model or strategy is deciding what to virtualize and what to expose. Successful virtualization preserves productivity, performance, and portability. Vectorizing compilers were successful at virtualizing the vector instruction set; although they exposed the *presence* of vector instructions, they virtualized the details of the instruction set, such as the vector register length, instruction spellings, and so on. Vectorizing compilers are still in use today, generating code for the x86 SIMD (SSE and AVX) and Power Altivec instructions. There are other ways to generate these instructions, such as the Intel SSE Intrinsics, but these can hardly be said to preserve productivity, and certainly do not promote portability. You might also use a set of vector library routines, such as the BLAS library, or C++ vector operations in the STL, but these don't compose well, and can easily become memory-bandwidth bound.

Another alternative is vector or array extensions to the language, such as Fortran array assignments or Intel's Array Notation for C. However, while these allow the compiler to more easily generate vector code, it doesn't mean it's better code than an explicit loop that gets vectorized. For example, compiling and vectorizing the following loop for SSE:

```
do i = 1, n
  x = a(i) + b(i)
  c(i) = exp(x) + 1/x
enddo
```

can be done by loading four elements of *a* and *b* into SSE registers, adding them, dividing into one, calling a vector *exp* routine, adding that result to the divide result, and storing those four elements into *c*. The intermediate result *x* never gets stored. The equivalent array code would be:

```
x(:) = a(:) + b(:)
c(:) = exp(x(:)) + 1/x(:)
```

The array assignments simplify the analysis to determine whether vector code can be generated, but the compiler has to do effectively the same amount of work to generate efficient code. In Fortran and Intel C, these array assignments are defined as computing the whole right hand side, then doing all the stores. The effect is as if the code were written as:

```
forall(i=1:n) temp(i) = a(i) + b(i)
forall(i=1:n) x(i) = temp(i)
forall(i=1:n) temp(i) = exp(x(i)) + 1/x(i)
forall(i=1:n) c(i) = temp(i)
```

where the *temp* array is allocated and managed by the compiler. The first analysis is to determine whether the *temp* array can be discarded. In some cases it cannot (such as a(2:n-1) = a(1:n-2) + a(3:n)), and the analysis to determine this is effectively the same dependence analysis as to vectorize the loop. If successful, the compiler is left with:

```
forall(i=1:n) x(i) = a(i) + b(i)
forall(i=1:n) c(i) = exp(x(i)) + 1/x(i)
```

Then the compiler needs to determine whether it can fuse these two loops. The advantage of fusing is avoiding the reload of the SSE register holding the intermediate value *x*. This analysis is essentially the same as for discarding the *temp* array above. Assuming that's successful, we get:

```
forall(i=1:n)
  x(i) = a(i) + b(i)
  c(i) = exp(x(i)) + 1/x(i)
endforall
```

Now we want the compiler to determine whether the array *x* is needed at all. In the original loop, *x* was a scalar, and compiler lifetime analysis for scalars is precise. For arrays, it's much more difficult, and sometimes intractable. At best, the code generated from the array assignments is as good as that from the vectorized loop. More likely, it will generate more memory accesses, and for large datasets, more cache misses.

At the minimum, the programming model should virtualize those aspects that are different among target systems. For instance, high level languages virtualize instruction sets and registers. The compilers virtualize instruction-level parallelism by scheduling instructions automatically. Operating systems virtualize the fixed physical memory size of the system with virtual memory, and virtualize the effect of multiple users by time slicing.

The model has to strike a balance between virtualizing a feature and perhaps losing performance or losing the ability to tune for that feature, versus exposing that feature and improving potential performance at the cost of productivity. For instance, one way to manage separate physical memories is to essentially emulate shared memory by utilizing virtual memory hardware, using demand paging to move data as needed from one physical memory to another. Where it works, it completely hides the separate memories, but it also makes it hard to optimize your program for separate memories, in part because the memory sharing is done at a hardware-defined granularity (virtual memory page) instead of an application-defined granularity.

Grab your Machete and Pith Helmet

If parallel programming is hard, heterogeneous programming is that hard, squared. Defining and building a productive, performance-portable heterogeneous programming system is hard. There are several current programming strategies that attempt to solve this problem, including OpenCL, Microsoft C++AMP, Google Renderscript, Intel's proposed offload directives (see slide 24), and the recent OpenACC specification. We might

also learn something from embedded system programming, which has had to deal with heterogeneous systems for many years. My next article will whack through the underbrush to expose each of these programming strategies in turn, presenting advantages and disadvantages relative to the goal.

About the Author

Michael Wolfe has developed compilers for over 30 years in both academia and industry, and is now a senior compiler engineer at The Portland Group, Inc. (www.pgroup.com), a wholly-owned subsidiary of STMicroelectronics, Inc. The opinions stated here are those of the author, and do not represent opinions of The Portland Group, Inc. or STMicroelectronics, Inc.
