A Strategy for Automatically Generating High Performance CUDA Code for a GPU Accelerator from a Specialized Fortran Code Expression

Pei-Hung Lin, Jagan Jayaraj, and Paul R. Woodward
Laboratory for Computational Science & Engineering
University of Minnesota

***Abstract*— Recent microprocessor designs concentrate upon adding cores rather than increasing clock speeds in order to achieve enhanced performance. As a result, in the last few years computational accelerators featuring many cores per chip have begun to appear in high performance scientific computing systems. The IBM Cell processor, with its 9 heterogeneous cores, was the first such accelerator to appear in a large scientific computing system, the Roadrunner machine at Los Alamos. Still more highly parallel graphic processing units (GPUs) are now beginning to be adopted as accelerators for scientific applications. For GPUs the potential performance is even greater than for Cell, but, as for Cell, there are multiple hurdles that must be surmounted in order to achieve this performance. We can think of multicore CPUs, like the Intel Nehalem-EX and IBM Power-7, as on one end of a spectrum defined by a large ratio of on-chip memory relative to peak processing power. The Cell processor has a smaller value of this ratio, while GPUs have still much smaller values. As one goes from large to small values of this ratio, in our view, the programming effort required to achieve a substantial fraction of the potential peak performance becomes greater. Our team has had good experience in meeting the programming challenges of the Cell processor [1-3], and we have developed a strategy for extending that work to GPUs. Our plan is to produce an automated code translator, as we have done for Cell, in order to reduce the GPU programming burden for scientific applications. In this paper we outline our strategy and give some very preliminary results.**

Keywords – GPU, code generation, memory management, performance optimization.

I. Viewing the GPU from a Vector Programmer's Perspective

For high performance scientific computing, we view the key ingredient of standard multicore CPUs, of the IBM Cell processor, and of the Nvidia Tesla or Fermi GPU as an n-way SIMD processor that operates on either 4- or 8-word operands, together with a capability to pipeline these operations. Thus we view these central computing engines as vector processing engines. The HPC community has decades of experience with classic Cray-style vector engines, but this community has had difficulty adapting to standard microprocessor CPUs despite their key similarity to the older Crays. We believe that the principle reason for this difficulty is the fact that all of the above mentioned modern computing engines have less memory bandwidth relative to their peak computing rates than the classic Crays by an order of magnitude or more. In these same relative terms, the Cray-1 had 3 times less memory bandwidth than the earlier Star-100, but it compensated for this deficiency by adding 8 vector registers. These registers permitted reduction of the demand for memory bandwidth by allowing intermediate results to be temporarily stored close to the processor. Today's CPUs utilize this same strategy by adding still more registers (the Cell SPU has 128 and the Nvidia GPU's SM, or streaming multiprocessor, has 64) plus some amount of on-chip cache memory. Today's processors form a spectrum which we can parameterize principally in terms of the amount of on-chip storage (since the numbers of registers are roughly comparable).

The Nvidia C1060 Tesla GPU contains 30 SMs. We view each SM as a vector processor. As with Cell's SPUs or the cores on an Intel or IBM standard CPU, we write a program to execute on the SM, and then we write a calling program that invokes several simultaneous instantiations of this program to execute independently in parallel. This simultaneous execution is no special consideration, as it is familiar, for example, from programs written for multiple vector engines as long ago as the mid 1980s. Our program for the SM will be composed of vectorizable loops. We are writing an automated code translator, as we did for Cell, to transform this vector code into the GPU's thread-based paradigm. Here the key point is to enable programmers to use a standard and familiar coding style in place of Nvidia's apparently quite different style, as encapsulated in the CUDA programming language. However, simple translation of familiar vector code into CUDA is not enough. We must also enable the programmer to manage effectively the precious, tiny on-chip data storage resources of the GPU's SM. Our experience with the Cell SPU is helpful here, because the SPU presents a similar challenge, although with about an order of magnitude more on-chip storage. Nevertheless, the memory bandwidth deficiencies of the SPU and the SM are nearly equal, so we can expect a similar strategy to be successful on both. To quantify this comparison, the Cray-1 memory could

This work has been supported through grants CNS-0708822 and OCI-0832618 from the National Science Foundation, and our code development has been supported by the Department of Energy through a contract from the Los Alamos National Laboratory.

provide 1 vector load or store for every 2 peak vector flops. The SPU's device memory delivers 1/16 of this bandwidth, and Nvidia's Tesla SM delivers 1/12.2 of this. This comparison does not, however, introduce the parameter of vector length. Good performance on the SM requires a vector length (threads per thread block, to a CUDA programmer) of 64, which is 8 times its SIMD processing width. In addition, performance demands that at least 3 of these vector programs execute on each SM simultaneously. This simultaneous execution of identical vector programs allows the SM to switch contexts when execution in one such program stalls, and thus to keep the SIMD engine busy. This concept is essentially the same as simultaneous multi-threading on standard CPUs. The need for these simultaneous executions puts considerable pressure on the on-chip data storage resources. We can see this if we think of the on-chip store as holding some number of "vector registers." Our programs for the Cell SPU treat the local store in this way. The vector length we find to deliver good performance on the SPU is 16, and we have roughly 100 of the 256 KB of storage left over after placing the program in the other 156 KB. This gives us room for 1600 vector register equivalents. Given the Tesla SM's smaller 16 KB of "shared memory" and 16 KB of register space, and given the need for vector lengths of 64 with 3 simultaneous instantiations, we have for each of the 3 vector programs space for only about 40 vector register equivalents. This is 40 times less space than we find in the SPU's local store for temporary data. In order, relatively speaking, to approach Cray-1 levels of performance on these GPU devices, we must therefore find ways to exploit this small on-chip storage.

Viewing the Cell SPU in this fashion, we have been able to implement our hydrodynamic codes so that they achieve about 1/3 of peak (7.7 Gflop/s/SPU), which is close to Cray-1 performance in the measure discussed above. The SPU performance falls by about 16% (to 6.3 Gflop/s/SPU) when it must obtain its data from its own main memory rather than exclusively from its local store, and falls by another 29% (to 4.5 Gflop/s/SPU) when we change the algorithm so that it demands roughly 3 times the data from main memory but only performs about 50% more work. Effective Cell performance falls by another 15% to 25% when we use hundreds (3.9 Gflop/s/SPU) or thousands (3.4 Gflop/s/SPU) of Cell processors in a large IBM Roadrunner system.

II. A GPU Programming Strategy

On the Cell SPU, we construct our program as follows: (1) we prefetch a small block of data to process, (2) we unpack the data to produce a number of short vectors that reside in the on-chip memory, (3) we update the data via many vector floating point operations, (4) we repack the updated data into a new data record, and (5) we write the new record back to main memory. This same procedure should work equally well on the GPU's SM, except that its on-chip memory may not be able to accommodate all the vector temporaries involved in the computation in step 3. We may also need to read the data record in stages, to save on-chip space. Our program will manage the on-chip space for registers and for "shared memory" vectors. Our Cell SPU programs must be modified so that they essentially use only the register file and do not rely upon the local store for data. To make this modification, we need to take control within our programs of the allocation of this precious on-chip "register" storage.

At first glance, we might think that we need only convert our Cell SPU program into CUDA syntax. We could let the CUDA compiler allocate our register storage for us in this model. In this case, we would declare all vectors in the SPU's local store to be register variables in CUDA (in "local storage"). However, were we to do this, we would be unable to exploit the GPU SM's 16 KB of "shared memory." Also, the shared memory must be used if we are to index our on-chip vectors with offsets, in order to evaluate difference equations. Our strategy therefore becomes one in which we explicitly map our vector temporaries to two sets of effective register variables. One set, which we might call r01, r02, . . . , will be in CUDA register variables; the other set, say s01, s02, etc., will be in shared memory. Our code must move temporaries from one memory set to the other as needed, and it must spill these variables to device memory (global memory) and retrieve them as needed. We believe that it will be nontrivial to rearrange the sequence of our Cell SPU vector instructions in such a way that the register storage space of the GPU SM is optimally exploited. This is one potential service that the translator could provide. After our program is converted to use the 40 or so different register variables exclusively, we note that it will become entirely unreadable. It is therefore essential that this code transformation be performed by an automatic code translation tool that has been thoroughly debugged. Once this code transformation is accomplished, we believe that the result will be better than any automatic caching of the data in hardware, since no unnecessary write backs of registers to the device memory will occur.

III. Preliminary Results

We have implemented one example code fragment on the GPU, and we are implementing a second, simplified but entire fluid dynamics code. The code fragment we implemented is taken from the original PPM code and represents roughly half the flops in single-fluid PPM. It was initially implemented in Fortran in Cell-processor style as a sequence of 64-way SIMD vector operations. In this Fortran form, it achieves 11.7 Gflop/s/core (50% of peak) on the Intel Nehalem quad-core processor (2.93 GHz). When it is implemented with only 64 threads per SM, it achieves 49 Gflop/s. When 3 of these thread blocks are executes simultaneously on each SM, each thread block takes 21 registers/thread and 4660 bytes of shared memory, and achieves 100 Gflop/s. If the entire code were to run at 100 Gflop/s on the GPU, then it would exceed Nehalem quad-core CPU performance by a factor of 3 to 4. A full Godunov example code has been excerpted from our present multi-fluid PPM code. It represents roughly 20% of the flops in the two-fluid PPM code. It runs at 7.6 Gflop/s/core (32% of peak) on the Intel Nehalem processor (2.93 GHz). We are implementing the Godunov hydrodynamics example on the GPU with the above programming strategy, and we will present the results at the conference. We hope to achieve the same 100 Gflop/s performance of the previous example on Tesla.

IV. Conclusion

We have drawn a parallel between the Cell processor and the GPU which suggests a strategy for extending to GPUs our successes in code conversion for Cell. Because of the much smaller on-chip memory on the GPU relative to its potential computational capability, we will need to devise a strategy for a form of register allocation and also for code rearrangement that minimizes the number of registers used. The code translation tools we have built for Cell and our experience in restructuring our codes for that platform will, we expect, mean that much of the most difficult code conversion will already have been automated. Our plan is to add to our code translation tools a further capability to support the additional register allocation code transformations and code rearrangement that our strategy for the GPU platform requires.

References

[1] P. R. Woodward, J. Jayaraj, P.-H. Lin, and P.-C. Yew, Moving scientific codes to multicore microprocessor CPUs, Computing in Science and Engineering, Nov. 2008, 16-25.

[2] P. R. Woodward, J. Jayaraj, P.-H. Lin, and W. Dai, First experience of compressible gas dynamics simulation on the Los Alamos Roadrunner machine. Concurrency and Computation: Practice and Experience, 2009.

[3] P. R. Woodward, J. Jayaraj, P.-H. Lin, and D. Porter, Programming techniques for moving scientific simulation codes to Roadrunner. Tutorial given 3/12/09 at Los Alamos. Available at www.lanl.gov/roadrunner/rrtechnicalseminars2008.