An Accelerator Programming Model for Multicore

Brent Leback, Steven Nakamoto, *and* **Michael Wolfe**,
The Portland Group (PGI)

ABSTRACT: *We have previously introduced the kernels programming model for accelerators such as GPUs, where a kernel roughly corresponds to tightly nested parallel loops with rectangular limits. We have designed directives for C and Fortran programs to target this model, similar in design to the well-known and widely used OpenMP parallel directives, and are implementing the directives and program model in the PGI C and Fortran compilers. This paper explores using the kernels programming model and directives on a multicore target. We attempt to answer two questions: whether the model can target a multicore processor, and whether it's a good idea.*

The paper briefly reviews the kernels programming model, where a kernel can include both MIMD (doall) and SIMD (vector) parallelism, explicitly or implicitly. The paper describes the directives used to program to the model, focusing on those important for a multicore target. The compiler implementation is presented in some detail, including the required analysis and code generation; differences from the implementation for an accelerator target are highlighted. An important feature of the implementation is the compiler feedback, which tells users the details of the generated code, allowing them to tune their program. The final section evaluates the model, comparing it to other parallel programming models such as OpenMP and automatic parallelization and vectorization.

KEYWORDS: Compiler, Accelerator, Multicore, Parallelization, Vectorization

1. Introduction

We can break current methods to program multicore processors into five broad categories:

• Use a parallel-optimized library. This can be and has been a very successful way to take advantage of any new architectural features. This was the motivation behind the Basic Linear Algebra Subroutine (BLAS) library, for instance. If the library is ported and optimized for a new machine with new features, then (hopefully) any application written using the library will get the benefit of those features with no more work than perhaps rebuilding with the new library.

• Use a parallelizing compiler. This has been the dream of many compiler research groups for well over 30 years. Great strides have been made in this realm, though it's far from solving the whole problem.

• Use a parallel language (such as Cilk) or parallel extensions to an existing language (OpenMP). This allows the programmer to expose parallelism beyond what a compiler can (re)discover, and perhaps with more granularity or locality control than is available using a library.

• Use MPI. This ignores the multicore aspect of the processor, treating it like a network of nodes. This has the advantage that the program will port to clusters and larger machines, but it doesn't use the multicore as efficiently as possible.

• Use low level parallelism, such as POSIX threads, directly. This allows very efficient thread control and synchronization, but can require truly heroic programming effort for larger applications.

In this paper, we discuss (yet) another method. Previously, we described the *kernels* programming model, targeting accelerators in general, and GPUs in particular[19]. A *kernel* roughly corresponds to a tightly nested loop. We described directive-based extensions to C and Fortran to target this model, where the directives are similar in design and scope to the OpenMP directives. Here, we discuss using the kernels programming model to program multicore processors, similar to treating the

multicore as an accelerator. The goal of the work described in this paper is to explore whether the kernels programming model can target a multicore processor and how to implement a compiler to do so, and whether such an effort is worthwhile.

We start by describing the goals that drove us to develop our programming model and directives in Section 2; we look at the success of the dominant method to program vector computers over the past 30 years. The kernels programming model is described in Section 3; the model is clearly inspired by today's GPUs, but there are several general concepts that map well onto multicores. Section 4 briefly describes our directives, focusing on those relevant for multicore targets. We give a possible compiler implementation in Section 5, comparing it to autoparallelizing compilers, OpenMP compilers, and our accelerator compiler implementation. An important feature of this programming method is effective compiler feedback; we discuss this in Section 6. We evaluate our proposed design and implementation against other methods and against our goals in Section 7. We conclude by discussing how we think the model and implementation will evolve.

2. Vector Programming

The first vector computers were the Texas Instrument's Advanced Scientific Computer (ASC), Control Data STAR100, and Burroughs Illiac IV, in the late 1960s and early 1970s. The TI ASC came with an aggressive vectorizing Fortran compiler[17]. The first commercially successful vector computer was the Cray-1; the primary programming mechanism was its vectorizing compiler, the Cray Fortran Translator (CFT). Here we focus on the success of vectorizing compilers, and CFT in particular.

The Cray-1 was not only a very fast vector processor; a great deal of its success was because it was the fastest scalar processor in the world at the time. With an 80MHz clock, it was twice the speed of the Control Data 7600, the machine it was to replace in many installations; many programs immediately ran quite a bit faster on the Cray. However, certain loops could achieve another factor of 5-10 speedup by using the vector instruction set.

Some loops would vectorize and achieve this speedup right away; most of the time, some programmer intervention was required. Even so, using the vectorizing compiler had some significant advantages over other vector programming styles.

- No new language was needed. The only extensions were a few built-in routines and some vectorization directives (such as CDIR$ IVDEP). The modified program could usually still be compiled and tested or run on workstations or other computers.
- It was incremental. Only innermost loops needed to be modified or rewritten to vectorize. The analysis was usually limited to that loop, or perhaps that routine. Each loop could be analyzed and rewritten in isolation, without coordinating as to how the rest of the program was being optimized.
- The compiler gave feedback as to its success or failure in vectorizing loops. In particular, it could be very precise about what statement, or what array reference in which statement prevented vectorization. This allowed programmers to focus their modifications to get the desired vector performance.
- The programming style was portable as well. As vector processors were developed by Convex, Fujitsu, Hitachi, IBM, NEC, and others, programs that would vectorize for the Cray would also vectorize for these other machines.

Successful vector programming required work from both the programmer and the compiler; essentially, the optimization effort was divided between the application developer and the compiler writer. Because of the vectorizing compiler, the application developer didn't have to learn a new language or maintain several versions of the program, and didn't have to dive into assembly language. The compiler writer could focus on getting good performance from the vectorizable subset of the language; the compiler didn't have to attempt heroic whole program analysis because it could always give up and tell the programmer where it needed help. The work division and cooperation was then and continues today to be successful. We designed our kernels programming model to benefit from the four advantages listed above.

3. The Kernels Model

Many parallel programming models have been proposed and implemented. A *threads* model creates a number of processing threads, where each thread has some private memory and all threads share a global memory; through structured or unstructured synchronization, the threads can cooperate on a parallel program. Cooperating Sequential Processes (CSP)[5] is a threads model. OpenMP[14] currently uses a threads model, based loosely on POSIX threads[4].

A *tasks* model creates a number of tasks in a task queue or container; some number of actors (such as threads) dequeue or remove a task and execute it, which may add more tasks to the queue or container. Some models have multiple or nested queues. Cilk[3] uses a

tasks model.

High Performance Fortran[10] was designed to use an *implicit parallelism* model with distributed data; conceptually, there was one thread of computation implemented with multiple processes executing on many processors, with redundant execution on replicated data and parallel execution on distributed data.

Here we describe what we call the *kernels* programming model. In this model, a program is a sequence of parallel kernels launched or invoked by a master thread. Each kernel is invoked and executed in parallel on a multidimensional domain. Essentially, the kernel is a multidimensional parallel loop, with the body of the loop comprising the kernel code, and the parallel loops describing the domain. The model allows for two types of parallelism; in the domain, each dimension can be designated as either a MIMD or a SIMD dimension. These can be modeled by two types of parallel loop: *doparallel* (MIMD) and *dovector* (SIMD). No synchronization is supported or allowed between kernel instances executing across different indices in a *doparallel* dimension. Kernel instances with the same *doparallel* indices but with different *dovector* indices can synchronize at a *barrier*[9]; this may be required for an accelerator, but is not required for multicore. These groups of kernel instances will have the same rank and size.

Each kernel is executed to completion before the next kernel is initiated; parallelism is exploited between the iterations of each kernel, not between multiple kernels. Using an accelerator, the host thread can execute asynchronously in parallel with the kernels, or to wait for some particular kernel to complete. On a multicore, it is expected the host thread takes part in the kernel computation.

The kernels parallelism model is clearly inspired by GPUs, CUDA[13], and OpenCL[12]. For NVIDIA GPU targets, *dovector* loops map to a thread block, and the *doparallel* loops map to the grid of thread blocks. We separate the model from the implementation so we can develop a programming style that targets the model, and a compiler that starts from the model to compile down to the target accelerator.

The architectural concepts required to support the kernels model are quite common. In particular, the two levels of parallelism, MIMD and SIMD, map pretty directly onto multicore (MIMD) processors with packed (SIMD) instructions. The compiler can map *doparallel* loops onto OpenMP style parallelism, and *dovector* loops onto the packed or SIMD instructions of a single core.

An important characteristic of accelerators is that they can perform fast context switches, which means for peak performance it is important to generate many more doparallel iterations than there are actual hardware processing elements. In this way, memory latency can be hidden, and the effective memory bandwidth increased.

In the future, if and when hyperthreading technologies mature, oversubscribing X64 cores using this model may potentially be a way to optimize for worsening memory bandwidth issues that were described last year[20].

4. The Directives

OpenMP serves as a higher level programming style for threads programming than POSIX threads[4]. OpenMP has not replaced pthreads, but is sufficiently expressive and efficient for many applications, and is more accessible for most developers. We designed a directive-based programming language to do for accelerator computing (exemplified by CUDA and OpenCL) what OpenMP has done for threads programming. We use directive syntax similar to OpenMP, in that we use a *sentinel* (stylized comment) in Fortran and `#pragma` syntax in C.

We propose two basic directives. The first defines a region of code containing loops, where each inner loop body is intended to map onto a kernel and the loop iterations map to the kernel domain. This directive can have additional clauses to describe the data locality and access restrictions. The accelerator region is delimited in Fortran by `region` and `end region` directives, as:

```
!$acc region
  ...
!$acc end region
```

In C we use a `region` pragma immediately followed by a
structured block.

```
#pragma acc region
{
  ...
}
```

Optional clauses on the region directive can include `copyin` or `copyout`, which name arrays or array sections that should be allocated on the accelerator and uploaded from or downloaded to the host; these are not relevant on a multicore, where all the cores share global memory.

We use another directive to describe the mapping of the program loop-level parallelism; specifically, the model provides MIMD parallelism (called *doparallel* here) and SIMD parallelism (called *dovector*). A programmer can map a loop to parallel mode or vector mode, or specify

that a loop should be strip-mined[2] with different execution modes for the resulting outer and inner loops. We show by example:

```
! Example 1:
!$acc region
 !$acc do parallel
 do j = 2,m-1
  !$acc do vector
  do i = 2,n-1
   r(i-1,j-1) = 0.25*(s(i-1,j) + s(i+1,j) &
                    + s(i,j-1) + s(i,j+1))
  enddo
 enddo
!$acc end region
```

Here the do directives tell the compiler to run the outer loop in parallel and the inner loop in vector mode. Each vector of m −2 elements requires 3 ×m−4 elements of the input matrix s and m − 2 stores of r for 4 × (m − 2) operations, giving a *compute intensity* (operations per operand) of about 1.

A more data-efficient or cache-efficient approach is to tile the loops; this is specified using our directives as:

```
! Example 2:
!$acc region copyin(s(1:n,1:m)) copyout(r)
 !$acc do parallel,vector(8)
 do j = 2,m-1
  !$acc do parallel,vector(8)
  do i = 2,n-1
   r(i-1,j-1) = 0.25*(s(i-1,j) + s(i+1,j) &
                    + s(i,j-1) + s(i,j+1))
  enddo
 enddo
!$acc end region
```

The do directives here tells the compiler to strip-mine each loop with the inner loops running in vector mode on an 8 × 8 tile, and the outer loops running in parallel. Each tile requires a 10 × 10 submatrix of s and 8 × 8 stores for r for 4 × 8 × 8 operations, for a compute intensity of 256/164 or about 1.5, which should make more efficient use of the cache memory.

Other loop mapping clauses specify sequential execution within a kernel. In the absence of loop-mapping clauses, the compiler will analyze the parallelism and data access patterns to determine a mapping.

A third, optional set of directives and clauses allow the user to have finer control over data locality. This may be either to specify resident data across kernel invocations, or hints on use of different levels of the memory hierarchy within a kernel. As an example of the first case, consider the above code with another array, which is resident on the accelerator:

```
! Example 3:
!$acc device data(t)

!$acc region
!$acc do parallel
 do j = 2,m-1
  !$acc do vector
  do i = 2,n-1
   r(i-1,j-1) = 2.0 * t(i,j)   -          &
                0.25*(s(i-1,j) + s(i+1,j) &
                    + s(i,j-1) + s(i,j+1))
  enddo
 enddo
!$acc end region

!$acc region
   < another kernel that uses t >
!$acc end region

!$acc end device data region
```

Targeting an accelerator, there are important decisions to be made between the compiler and user regarding data which is to stay resident on the device through kernel invocations, data which is to be moved to and from the device at each invocation, data that is private, cached, shared, etc. Similarly, we have shown in previous papers[20] that even on x64 multicore systems, large performance differences can result from a compiler and/or user knowing where data resides, and using the proper prefetch, load, and store instructions which optimize for the best data transfer efficiencies. At this point, it is unclear whether there will be one general model a user can adhere to which can cleanly specify the distance between the expected data location at kernel invocation and the actual processing elements, and that proves effective on all targets.

The programming model and the directives to implement them are still evolving; more details are available at the PGI website www.pgroup.com/accelerate.

5. Compiler Implementation

We are implementing the kernels programming model using the proposed directives in the PGI Fortran (PGFortran) and PGI C (PGCC) compilers. Initially, these compilers target 64-bit x86 processors with an attached NVIDIA GPU or Tesla card[19]. This section describes a design for an implementation to target a multicore processor.

The steps within the compiler to analyze an accelerator region for a multicore are:

- Identify linear induction variables and compute loop trip counts[6].
- Detect loop-level parallelism, using classical data dependence analysis.
- Identify loop private scalars, using classical data-flow analysis.
- Identify loop private arrays, using array section analysis.

• Map or schedule the loop level parallelism onto the multicore processor.

• Compile parallel loops as if using OpenMP or auto-detected parallelism.

• Compile inner vector loops for the SSE instruction set.

• Add a barrier synchronization at the end of the accelerator region.

• Give feedback to the programmer with details about the generated code.

These steps can run as a phase fairly early in the compiler. Details follow, in particular comparing compiling for a multicore to compiling for an accelerator like a GPU.

5.1 Validation

When targeting an accelerator, the compiler must first validate the operations in the accelerator region as feasible on the target. For instance, some accelerators may only support single precision, or may not support the full math library. This is less of a concern when targeting a multicore, since each core is as capable as any other, and in particular as capable as the "host."

5.2 Parallelism Detection

The kernels model depends on loop-level multidimensional parallelism. We use classical data dependence analysis[1, 18] and parallelism identification. The analysis is augmented by user directives, and distinguishes between fully parallel (*doall*) and vector parallel loops, where a fully parallel loop has no loop-carried dependences and a vector parallel loop has no lexically-backward dependences.

On a multicore, vector operations are implemented with packed or vector instructions on a single core, so no synchronization is necessary. This contrasts with the explicit synchronization required between synchronous iteration thread groups on a GPU, for instance.

With both packed instruction sets and with today's accelerators, there is a big performance disadvantage if the data fetches and stores are not contiguous (stride-1) in memory. Also, in both cases there is a limited size to the vector iteration set. On the NVIDIA GPUs, the maximum size for a thread group (synchronous iteration group) is 256; the 256 threads in a thread group can be organized in a 1-D, 2-D or 3-D manner. On today's X64 multicore, the SSE instruction set has an effective vector length of 2 (double precision) or 4 (single precision); there is no hardware support for multidimensional vector operations. Software can emulate larger and multidimensional vector operations, or can simply scalarize the appropriate loops.

5.3 Scalar Analysis

We use classical scalar *def-use* and *live variable* analysis to find scalars live into and out of each loop. In particular, this allows the compiler to easily identify scalars that can be or should be *privatized*. For an accelerator, scalars live-in to the region and loop need to be identified to be explicitly copied over to the device memory; this is unnecessary for a multicore.

5.4 Array Section Analysis

We use a simple implementation of regular section analysis that finds rectangular array sections[8] used and modified in each statement and loop. This allows the compiler to *privatize* arrays or subarrays; a loop-private array is one where the addresses do not depend on the loop iteration, each array element fetch in the loop is dominated by an assignment to that element earlier in the loop, and the array is not used after the loop. The accelerator compiler uses array section analysis to manage data allocation on the device memory, and data movement between the device and the host, all of which is unnecessary on the multicore.

5.5 Parallelism Mapping

The parallelism mapping step is key to the final performance, and must be tuned for each target. For an accelerator, the mapping step is responsible for optimizing use of the limited resources, such as any software-managed cache memory or synchronous iteration thread group size. For a multi-core, the mapping step tries to find enough vector parallelism to utilize the vector instruction set effectively, and enough large-grain loop-level parallelism to implement efficiently with OS-managed threads. For accelerators, we have an initial implementation of an automatic mapper, which we call the *planner*, but in many cases efficient code requires users to insert mapping directives.

This step is key to meeting one of our hoped-for advantages; for the programming style to be considered portable, either the mapping must be automated and tuned for each target, or the same mapping directives must apply to different targets. We discuss this in more detail in Section 7.

5.6 Code Generation

Code generation for a multicore should be relatively easy, since this is built within a single-core compiler with OpenMP features. In particular, since there is only one instruction set, other program development tools (linker,

debugger) will work without customization.

The *doparallel* loops can be compiled into OpenMP-style parallel execution. For nested parallel loops, OpenMP uses either loop collapsing or nested parallel regions. Nested parallel regions have additional overhead, so we avoid them and use loop collapsing.

The *dovector* loops can be compiled into vector code for the SSE instructions, using existing compiler phases. The compiler need not emulate long vector operations; instead, it can optimize for register and memory locality, using loop unrolling or loop, choosing the most advantageous vector loop for actual SSE instructions.

6. Compiler Feedback

As we mentioned in Section 2, one of the keys to making mature vectorizing compilers successful in the past was the compiler feedback. The simplicity of the interface and division of work between tool and user enabled a generation of software engineers to become successful at performance tuning.

Today, most modern compilers discover and generate a wealth of information about what the compiler did in optimizing the code, what optimizations could not be implemented (and why), how data is accessed, relationships between procedures, and much more. PGI compilers include the ability to save this information into the compiled object and/or executable file for later extraction and review. The Common Compiler Feedback Format (CCFF) is a draft standard published by PGI that defines what compiler information is stored and how the information is formatted. Using CCFF, HPC tools providers can enhance their products to offer better information about optimizing performance.

Using PGI compilers, compiler feedback is generated by using the –Minfo option, and CCFF is generated by using –Minfo=ccff. CCFF information can be viewed along with a source browser and post-mortem performance data in the PGI PGPROF profiling tool. Plans are to eventually incorporate it into IDEs.

Given the previous example 2 loop:

```
! Example 2:
!$acc region copyin(s(1:n,1:m)) copyout(r)
 !$acc do parallel,vector(8)
 do j = 2,m-1
  !$acc do parallel,vector(8)
  do i = 2,n-1
   r(i-1,j-1) = 0.25*(s(i-1,j) + s(i+1,j) &
                    + s(i,j-1) + s(i,j+1))
  enddo
 enddo
!$acc end region
```

The compiler generates the following information:

```
10, Generating copyin(s(:n,:m))
    Generating copyout(r(1:n-2,1:m-2))
12, Loop is parallelizable
14, Loop is parallelizable
    Accelerator kernel generated
    12, !$acc do parallel, vector(8)
    14, !$acc do parallel, vector(8)
        Cached references to size [10x10]
           block of s
```

If we change some of the right-hand-side references of s, to instead access r in the vertical dimension, we can see that the compiler detects and reports dependencies in the loop:

```
12, Loop is parallelizable
14, Loop carried dependence of r prevents
      parallelization
    Loop carried backward dependence of r
      prevents vectorization
    Accelerator kernel generated
```

On an x86 platform, it is clear that users expect the compiler to continue and generate correct code even if it is unable to vectorize and/or parallelize the loops. On an accelerator, it is not so clear; if a directive is given to explicitly parallelize and vectorize a loop, and the compiler cannot do that due to either programmer error or compiler inadequacies, what is the right course of action? Should the compilation fail? Is compiler feedback enough? These are considerations we are currently working on with our early adopters. Currently, on an accelerator, for the above example, we will just generate a very inefficient kernel.

7. Evaluation

We evaluate our design by comparing it to OpenMP, to autoparallelism, to our goals, and to other portable approaches.

7.1 OpenMP

Does this programming model and compiler have any advantages or disadvantages compared to an equivalent OpenMP compiler? We see several differences:

- The OpenMP execution model is strictly threads-based, which has both advantages and disadvantages.
- OpenMP only addresses MIMD parallelism, leaving SIMD or vector parallelism to the programmer and compiler.
- OpenMP is much more general than the kernels model.
- OpenMP doesn't apply to accelerators.

We address these in more detail below.

OpenMP parallelism is defined in terms of threads. A certain number of threads are created, and some or all of these threads are active during parallel execution. Tasks, such as loop iterations or groups of loop iterations, are divided among the threads. In some cases, the mapping is explicit, such as the OpenMP `static` loop schedule, which maps a well-defined subset of iterations to each of the threads in a particular order. Such a mapping has the advantage of allowing a user to optimize for locality between parallel regions or between parallel loops, by scheduling loop iterations that access the same data onto the same thread. Unless the operating system reschedules that thread onto a different core or processor, the program can benefit from cache locality between parallel loops or regions. This is pretty low-level programming, and while it can be quite effective, it requires more changes than just adding loop directives. For instance, if we take our Example 2 program, the tiled Jacobi iteration, and implement it in OpenMP, the resulting program would require the tiling to be explicit:

```
! Example 2 - OpenMP:
!$omp parallel
!$omp do collapse(2)
do jt = 2,m-1,8
 do it = 2,n-1,8
  do j = jt,min(m-1,jt+7)
   do i = it,min(n-1,it+7)
     r(i-1,j-1) = 0.25*(s(i-1,j)+s(i+1,j)&
                       + s(i,j-1)+s(i,j+1))
   enddo
  enddo
 enddo
enddo
!$omp end parallel
```

Vector code generation for the inner loop is implicit here. If we want multiple levels of tiling, we must add another pair of loops.

Another disadvantage is that the programmer becomes a thread manager; he or she must be aware that there is a mapping between tasks and threads, and make sure to create enough threads to fill the available cores, but not so many as to create thrashing. Tuning an OpenMP program may require knowledge of the system parameters or the OpenMP vendor or implementation. While OpenMP programs are portable, a single program may need to be tuned differently for different systems, and there is no standard way to specify system-dependent tuning parameters, such as parallel loop schedules.

The kernels model is defined in terms of parallel loop iterations, essentially parallel tasks. In fact, our implementation will use OpenMP-style threads to execute the iterations, but the mapping between iterations and threads is much less strict. Thread management is handled by the compiler and its runtime library. The kernels directives also allow specification of tiled algorithms without changing the text of the program, and specifies both MIMD and SIMD parallelism in a single model. The kernels model also has tuning parameters, and the values may depend on the target accelerator; we return to this point in our concluding remarks.

OpenMP is much more general than the kernels model, in that more parallel programs can be written in OpenMP than can be written in the kernels model. While OpenMP began as a way to standardize the concept of a parallel loop, it also allows parallel sections, and now has dynamic parallel tasks. The kernels model only handles nested parallel loops; our goal is to focus on this important subset of parallel programs and implement it well across a range of targets.

Finally, OpenMP does not apply on today's accelerators. Because OpenMP is strictly shared-memory, thread-based, with the master thread participating in the parallel computation. With OpenMP, it is no longer the case that the sequential program (ignoring the OpenMP directives) is semantically the same as (gives the same answers as) the parallel program (respecting the OpenMP directives). Today's accelerators do not support shared memory with the host, do not support full thread synchronization even among threads on the accelerator, and do not allow participation by the master (host) thread. Research efforts to port OpenMP programs to accelerators only succeed when they can limit the OpenMP programs, ignore some directives, or deliver unacceptable performance penalties.

In summary, the kernels model is related to OpenMP, and will use some of the same low-level implementation details. The kernels model is more restricted in the types of parallel constructs it allows, but it has advantages in managing MIMD and SIMD parallelism and cache locality through tiling in a single mechanism.

7.2 Autoparallelism

Parts of the implementation are built on automatic parallelization, such as induction variable identification, dependence analysis, and loop-level parallelism detection. Could this entire process be automated, making the directives unnecessary?

In the best world, the directives would be unnecessary; the compiler would detect and map the parallelism onto the hardware without any need for user directives. However, the compile-time cost of the deep analysis needed to make this efficient, and the potential for a compiler to make bad choices, lead us to believe that user-directed parallelization is more likely to lead to

improved performance than depending entirely on automatic optimization. When successful, the benefit to parallel execution is high enough to warrant users spending more time in tuning. As discussed in Section 6, with performance feedback from the compiler, users can focus their effort where it is more likely to be effective.

In addition, users may have to recast their algorithms to take advantage of parallelism, or to avoid memory latency or bandwidth issues or other pitfalls of the target machine. Programmers can change a program in ways that are far beyond what any compiler or tool can do, and we think we have a mechanism that helps them do just that.

7.3 Other Approaches

Other recent portable approaches to manage multicore and accelerator parallelism are:

- Use a parallel-optimized library.
- Use a run-time code generation scheme, a la Rapidmind.
- Use a compiler to generate multicore code for accelerator programs, such as MCUDA.

NVIDIA delivers a GPU-enabled version of the Basic Linear Algebra Subroutines (BLAS) with the CUDA SDK. It can deliver very high performance for those functions implemented in the library, and if your program consists of BLAS calls, you can use the CUDA BLAS to advantage. The advantage to a compiler-based approach would be the performance feedback, but a well-tuned library shouldn't need to give any feedback. Our approach is language-based, hence at a lower level than a library. In general, if you can use a well-tuned library, you should probably do so.

Another type of library is exemplified by the RapidMind platform[11]. This exposes a C++ class library interface defining data types and methods with which you rewrite your application. At runtime, the underlying platform will generate appropriate code for the particular high performance target on which your program is running, be it an accelerator, GPU, or multicore processor. Such a platform promises high portability after a one-time rewrite to use the class library. Our approach is directly embedded in the language and the compiler, and so has the advantage of allowing performance feedback and programming tuning. However, our approach lacks a runtime code generation and tuning component.

Another method is to start with the GPU kernel and to compile and optimize that for multicore execution, exemplified by the MCUDA project[16]. If successful, it would present a single framework for writing parallel programs for GPUs and for multicore processors. However, optimized GPU kernels written in CUDA or OpenCL for today's GPUs include many low-level details tuned for the specific GPU. It remains to be seen whether such an approach can successfully produce a model that can efficiently target both the multicore and GPU.

7.4 Our Goals

We had four goals, matching the four advantages that classical vectorizing compilers had listed in Section 2. How well did we satisfy our goals? To recap, the four goals were:

- Existing languages (extensions, directives allowed).
- Incremental.
- Compiler feedback.
- Portable across targets.

We designed the directives with the first goal in mind, and we believe the model satisfies the goal.

The second goal is easier to satisfy for a multicore target than for an accelerator, because of the shared memory. One of the keys to performance on the accelerator is to minimize data traffic between the host and the accelerator memory, which is communicated over a PCI DMA channel. This means that often the programmer will want to allocate memory on the accelerator memory and leave it there. This can have a global impact across the whole program, because the data that lives on the accelerator is not accessible from the host.

On a multicore, data doesn't move from one memory space to another. Adding kernels model directives can change the execution behavior in that region, but will not affect the rest of the program. In that sense, program development and optimization is incremental.

We also designed the compiler with feedback as an integral part. We firmly believe that if performance is important, the programmer must be *in the loop* during optimization. As much as we would all like to believe in a magic compiler that will generate optimal code through deep program analysis, we have no reason to believe that this is any more feasible today than it was thirty years ago.

The final goal is still open. We believe we can certainly compile an accelerator region for a multicore, given the design from Section 5. However, there is still the problem of tuning the parallelism mapping. If we can converge on automatic parallelism mapping (Section 5.5), then we can say that we have a portable model. However, the state today is that programmers must frequently insert directives to map loop parallelism to accelerator hardware parallelism; this mapping is unlikely to be optimal for a different accelerator, or for a multicore processor.

8. Concluding Remarks

From our design and analysis, we conclude that a compiler targeting multicore processors for accelerator regions is feasible with reasonable effort, when built on a compiler that already supports accelerator regions and supports parallelism (OpenMP) and vector operations (SSE).

A key to success will be whether we can satisfy our fourth goal, that of having a parallel programming model that is portable across a wide variety of parallel systems. In this case, that means portable across different multicore processors as well as accelerators. The model and the directive language are both still under active development, and this question is still open. At this time, target-specific tuning is frequently required, which is a hindrance to true portability. Our hope is that as we gain experience and the model and implementation matures, we can find a way to specify the parallelism and desired mapping in a way that abstracts enough about the program that the right information is available for each different target.

As mentioned, the model and directives are under active development. Currently, the model is limited in many ways; some of these limitations could be relaxed or extended. For instance, the model could allow for parallelism between kernels, or pipeline parallelism between loop iterations, or more explicit data locality specifications. We are also looking at what items would be required to generate tuned code for other targets, such as other GPUs, or specialized processors like STI Cell[7] or Intel Larrabee[15].

About the Authors

Brent Leback is an Engineering Manager for PGI. He has worked in various positions over the last 25 years in HPC customer support, math library development, applications engineering and consulting at QTC, Axian, PGI and STMicroelectronics. He can be reached by e-mail at *brent.leback@pgroup.com*.

Steven Nakamoto is the Compiler Architect at PGI. Prior to joining PGI in 1989, he worked in various software and compiler engineering positions at Floating Point Systems, BiiN, Mentor Graphics, and Honeywell Information Systems. He can be reached by e-mail at *steven.nakamoto@pgroup.com.*

Michael Wolfe joined PGI as a compiler engineer in 1996; he has worked on parallel compilers for over 30 years. He has published one textbook, *High Performance Compilers for Parallel Computing*, and a number of technical papers. He can be reached by e-mail at *michael.wolfe@pgroup.com*.

References

[1] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman, 2002.

[2] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *Computing Surveys* (Dec. 1994), 345–420.

[3] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing* (Aug. 1996), 55–69.

[4] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[5] DIJKSTRA, E. W. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, P. B. Hansen, Ed. Springer Verlag, New York, 2002.

[6] GERLEK, M. P., STOLTZ, E., AND WOLFE, M. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems 17*, 1 (Jan. 1995), 85–122.

[7] GSCHWIND, M. The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming 35*, 3 (June 2007).

[8] HAVLAK, P., AND KENNEDY, K. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991), 350–360.

[9] JORDAN, H. F. The Force. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. MIT Press, 1987, pp. 395–436.

[10] KOELBEL, C. H., LOVEMAN, D. B., SCHREIBER, R. S., STEELE JR., G. L., AND ZOSEL, M. E. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Massachchusetts, 1994. Scientific and Engineering Computation Series.

[11] MCCOOL, M. D. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference* (Santa Clara, Cal., Oct. 2006).

[12] NUNSHI, A., Ed. *The OpenCL Specification*. The Kronos Group, Dec. 2008.

[13] NVIDIA CORP. *NVIDIA CUDA Cuda Unified Device Architecture Reference Manual*, June 2008.

[14] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface*, 2008.

[15] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. In *International Conference on Computer Graphics and Interactive Techniques* (Aug. 2008), ACM Press.

[16] STRATTON, J. A., STONE, S. S., AND MEI W. HWU, W. MCUDA: An efficient implementation of CUDA kernels on multi-cores. IMPACT Technical Report IMPACT-0801, University of Illinois, Center for Reliable and High-Performance Computing, Apr. 2008.

[17] WEDEL, D. Fortran for the Texas Instruments ASC system. *SIGPLAN Notices 10*, 3 (Mar. 1975), 119–132.

[18] WOLFE, M. *High Performance Compilers for Parallel Computing*. Reading, Mass.: Addison-Wesley, 1996.

[19] WOLFE, M. Design and implementation of a high level programming model for GPUs. In *Workshop on Exploiting Parallelism using Hardware Assisted Methods* (Seattle, Wa., Mar. 2009).

[20] LEBACK, B., DOERFLER, D., and HEROUX, M., *Performance Analysis and Optimization of the Trilinos Epetra Package on the Quad-Core AMD Opteron Processor,* CUG 2008 Proceedings.