Performance Portability and OpenACC

Douglas Miles, David Norton *and* **Michael Wolfe**
PGI / NVIDIA

ABSTRACT: *Performance portability means a single program gives good performance across a variety of systems, without modifying the program. OpenACC is designed to offer performance portability across CPUs with SIMD extensions and accelerators based on GPU or many-core architectures. Using a sequence of examples, we explore the aspects of performance portability that are well-addressed by OpenACC itself and those that require underlying compiler optimization techniques. We introduce the concepts of forward and backward performance portability, where the former means legacy codes optimized for SIMD-capable CPUs can be compiled for optimal execution on accelerators and the latter means the opposite. The goal of an OpenACC compiler should be to provide both, and we uncover some interesting opportunities as we explore the concept of backward performance portability.*

KEYWORDS: Compiler, Accelerator, Multicore, GPGPU, Parallelization, Vectorization

1. Introduction

A computer program is *portable* if a single source code version can be compiled and executed across multiple types of computer systems. Impediments to portability include differing instruction sets, differing capacity (e.g. memory size) and differing functionality of operating systems. High-level programming languages are designed to overcome these impediments. These languages can be implemented via traditional compilers, interpreters, or just-in-time compilers. Libraries including pre-packaged functions or class definitions enhance the portability of high-level programming languages.

A computer program is *performance portable* if a single source code version can be compiled and executed with uniformly good performance across multiple types of computer systems. Impediments to performance portability are often related to system architecture – memory hierarchy, parallelism, vectors – but the design of programming languages and the inability of compilers to successfully map those programming languages to a given target are big factors as well.

In this paper, we define a program as being *forward performance portable* if it is written and optimized for a previous generation or style of processor, and can be compiled for execution with high performance on a newer generation or style of processor. Likewise, a *backward performance portable* program is one written and optimized for a newer generation or style of processor and which can be compiled for execution with high performance on a previous generation or style of processor.

2. Performance Portability Success Stories

The most classical example of successful performance portable programs is the set of programs written for vectorizing compilers. In 1976, the Cray-1 supercomputer was installed at Los Alamos Scientific Laboratory. The Cray Fortran Translator (CFT) would automatically vectorize loops, which could give a 4-5X performance boost over sequential execution. Moreover, the compiler would include vectorization feedback in the listing file, indicating to the programmer which loops failed to vectorize and why, thereby directing users on how to modify their programs to enhance vectorization. Over the next few years, Cray programmers were trained in how to write programs that would successfully vectorize. Other manufacturers introduced vector supercomputer computers over the next several years, such as the IBM Vector Facility, the NEC SX-1 and SX-2, the Convex C1, and others. Performance portability was demonstrated when programs written to vectorize with CFT for the Cray would also vectorize for these other vector computers.

A second set of performance portable programs includes those using MPI for parallel control and communication. MPI has become ubiquitous for implementing scalable, highly parallel programs. As such, vendors are motivated to design machines that execute these programs efficiently, even to providing tuned implementations of the MPI libraries.

These two examples show two very different methods to achieve performance portability. The first uses a hardware feature (vector instruction set), exposed to and exploited by programmers using optimizing compiler technology (vectorization), implemented by multiple vendors with enough similarities that programs optimized to execute in vector mode for any vendor will benefit on all such machines. The second uses a software specification (MPI) that motivates vendors to optimize the hardware and/or system software for such programs.

3. Performance Portability Between CPUs and GPUs

OpenACC is intended and designed to provide both portability and performance portability across most types of processors used in HPC today: multi-core CPUs with SIMD instructions, many-core processors like the Intel MIC, massively parallel stream-oriented GPUs, and heterogeneous configurations where a CPU is coupled to a MIC co-processor or GPU accelerator. There are several examples of OpenACC benchmarks and applications that display performance portability across multiple types of systems [Her12, LSG12, MFM13].

One of the challenges of enabling performance portability across CPUs and GPUs is to overcome basic design differences in the areas of memory and parallelism. CPUs have latency-optimized memory systems and rely on a few very fast cores with modest SIMD capability. To deliver high performance on these processors, work is distributed across the cores using MPI or SMP parallelization and then organized for efficient use of the SIMD hardware using vectorization. In a paper presented at CUG 2013 [CO13], an example loop from the weather application COSMO was shown as an example of this style of coding:

```
do k=2,Nz
   do ip=1,nproma
       c2=c1(ip,k)*a(ip,k-1)
       a(ip,k)=c2*a(ip,k-1)
   enddo
enddo
```

Example 1

The inner loop above is stride-1, and will be vectorized by most optimizing compilers to take advantage of SIMD capabilities on an x86 CPU. A similar formulation would be advantageous on Power CPUs with Altivec, or ARM CPUs with Neon extensions.

GPUs have stream-optimized memory systems and rely on very large numbers of slower cores that operate in sub-groups in a single-instruction multiple-thread (SIMT) fashion. Every core in a sub-group executes the same instruction at the same time, or no instruction at all. To deliver high performance on these processors, work is oversubscribed across the cores in a massively parallel fashion, and organized so that when the cores in a sub-group each issue a given memory operation, the operations collectively result in a single main memory fetch of a sequence of contiguous data elements. To structure the above loop from COSMO for optimal execution on a GPU, the loops must be interchanged:

```
do ip=1,nproma
   do k=2,Nz
      c2=c1(ip,k)*a(ip,k-1)
      a(ip,k)=c2*a(ip,k-1)
   enddo
enddo
```

Example 2

When compiled for execution on a GPU using either OpenACC or an explicit model such as CUDA or OpenCL, the outer ip-loop in Example 2 is replaced with a parallel kernel launch call and the inner loop becomes the body of the kernel. Each outer ip-loop iteration is executed by a separate GPU core, each of which executes separate complete instances of the inner k-loop. The work is scheduled by the compiler so that adjacent GPU cores tend to access adjacent elements of `c1` and `a` with the same memory access instruction, resulting in optimal use of memory bandwidth on the GPU.

The resulting problem is obvious. If we have to structure loops in one way for optimal execution on a CPU and in another way for optimal execution on a GPU, it puts at risk the ability to maintain a single version of source code that can be efficiently compiled and executed on either CPUs or GPUs. The above example is quite simple to conditionally compile either way, but the same situation can occur with loops that are much larger and more complicated. If this challenge can't be resolved by a programming model or compiler, the application developer must either maintain two code paths for computationally intensive code, or compromise the performance of one platform or the other.

Can we make the CPU formulation forward performance portable to GPUs? Can we make the GPU formulation backward performance portable to CPUs? These are the questions we will explore in the remainder of this paper.

4. A Motivating Example

FIM (Flow-following, finite-volume, icosahedral model) is a weather code under development by NOAA (http://fim.noaa.gov). To facilitate performance analysis and tuning, the authors of FIM created the standalone TRCADV benchmark representing a key part of the code. There are 3 subroutines in TRCADV. Example 3 shows the main loop structure of `trcadv3`.

```
!$acc parallel num_gangs(10242) &
!$acc vector_length(64) private(anti_tdcy)
!$acc loop gang
        do ipn=ips,ipe
!$acc loop vector
          do k=1,nvl
            anti_tdcy(k) = 0.
          end do
          do edg=1,nprox(ipn)
!$acc loop vector
            do k=1,nvl
              if (antiflx(k,edg,ipn) >= 0.) then
                 antiflx(k,edg,ipn) =         &
                 antiflx(k,edg,ipn)*          &
                 min(r_mnus(k,ipn),           &
                     r_plus(k,prox(edg,ipn)))
              else
                 antiflx(k,edg,ipn) =         &
                 antiflx(k,edg,ipn) *         &
                 min(r_plus(k,ipn),           &
                     r_mnus(k,prox(edg,ipn)))
              end if
              anti_tdcy(k) = anti_tdcy(k) +   &
                             antiflx(k,edg,ipn)
            end do
          end do
!$acc loop vector
          do k=1,nvl
            anti_tdcy(k) =                    &
               -anti_tdcy(k)* rarea(ipn)
            trc_tdcy(k,ipn,nf,t) =            &
               trclo_tdcy(k,ipn,nf,t)         &
             + anti_tdcy(k)
            trcdp(k,ipn,t) =                  &
               trcdp(k,ipn,t)                 &
             + adbash1*trc_tdcy(k,ipn, nf,t)  &
             + adbash2*trc_tdcy(k,ipn, of,t)  &
             + adbash3*trc_tdcy(k,ipn,vof,t)
            tracr(k,ipn,t) =                  &
               max(trmin(k,ipn),              &
                   min(trmax(k,ipn),          &
                       trcdp(k,ipn,t) /       &
                       max(thshld,delp(k,ipn))))
          end do
        end do
!$acc end parallel
```

Example 3

The nature of the computations is different in the other two subroutines (`trcadv1` and `trcadv2`), and they are somewhat larger, but the basic loop structure is identical across all three. The main loop is designed to be parallelized with either OpenMP or OpenACC. It is structured with modern CPU architectures in mind, with a parallelizable outermost ipn-loop and SIMD vectorizable innermost k-loops. It uses the OpenACC `parallel` construct, which is conceptually similar to an OpenMP parallel region and is frequently used to incrementally port codes from OpenMP to OpenACC [LL12].

Tables 1a and 1b show the increasing performance of the code on a single 3.2Ghz Sandy Bridge CPU using PGI 14.4 and Intel 14.0.2 compilers when SIMD vectorization and OpenMP parallelization are enabled, versus a serial non-vectorized version running on only 1 core. All times in microseconds.

Cores	trcadv1	trcadv2	trcadv3
1*	76100	62400	67400
1	44500	15900	60900
2	29800	9700	34100
4	23800	6900	17400
6	23600	6500	13100

*No SIMD vectorization

Table 1a – PGI 14.4 Fortran compiler

Cores	trcadv1	trcadv2	trcadv3
1*	74,300	62,000	56,300
1	48,600	18,900	31,100
2	30,100	10,600	16,500
4	26,100	6,900	10,000
6	26,800	6,800	8,600

*No SIMD vectorization

Table 1b – Intel 14.0.2 Fortran compiler

Table 2 shows the performance of the code on an NVIDIA Kepler K20x GPU, unmodified, using OpenACC and the PGI 14.4 compilers.

Cores	trcadv1	trcadv2	trcadv3
2688	1240	750	810

Table 2

The Kepler executes kernels using two levels of parallelism. The 14.4 version of the PGI compilers maps the outer loop gang parallelism to the Kepler thread blocks; the thread blocks must execute completely independently, since there is no support for synchronization between thread blocks or a barrier across thread blocks. The compiler maps the inner loop vector parallelism to the threads within a thread block; there is a barrier of the vector lanes at the end of a vector loop. The times in Table 2 do not include any data transfers, just the on-GPU compute times.

The performance is good relative to a single CPU, but feedback from NOAA indicated that the GPU performance should be up to 50% faster, based on timings using the F2C-ACC Fortran to CUDA C translator [Gov09].

4.1 Loop interchange

For a GPU, we want to push vector and parallel loops outward to minimize synchronization of the GPU cores and maximize the range of code over which coordinated memory accesses occur. This creates longer regions of code during which the GPU cores run freely at full speed and allows the GPU memory system to ramp up to and sustain a high level of memory bandwidth utilization.

For the loops in example 3, we can interchange the sequential `do edg` loop with the inner vector `do k` loop manually. This increases the work in each vector loop iteration and enables better optimization of the sequential inner loop. Example 4 shows the resulting loop structure:

```
!$acc loop vector
      do k=1,nvl
         do edg=1,nprox(ipn)
            if (antiflx(k,edg,ipn) >= 0.) then
               antiflx(k,edg,ipn) =         &
               antiflx(k,edg,ipn)*          &
               min(r_mnus(k,ipn),           &
               r_plus(k,prox(edg,ipn)))
            else
               antiflx(k,edg,ipn) =         &
               antiflx(k,edg,ipn) *         &
               min(r_plus(k,ipn),           &
               r_mnus(k,prox(edg,ipn)))
            end if
            anti_tdcy(k) = anti_tdcy(k) +   &
            antiflx(k,edg,ipn)
         end do
      end do
```

Example 4

For any such transformation to be done automatically by the compiler, it must answer two questions: Is it legal? Is it profitable? In the case shown here, determining legality of the loop interchange is quite trivial since the k loop has no cross-iteration dependences, so no dependence conditions prevent interchanging. However, when the expressions used in the loops become more complicated, legality checks can become difficult. If a compiler can't determine a transformation is legal, it must assume it is unsafe and no transformation can be performed unless some mechanism (like a directive) is used to convey to the compiler that it is safe.

Table 3 shows the performance of the code on an NVIDIA Kepler using OpenACC after this loop interchange is performed (V1), compared to the original version (V0).

Version	trcadv1	trcadv2	trcadv3
V0	1240	750	810
V1	1060	610	730

Table 3

Clearly it is profitable for the GPU in this case. We expect that such interchange, designed to drive vectorizable loops outward, is almost always profitable for a GPU. The `do k` loop must be strip-mined to allow for arbitrary values of `nvl`, and the interchange has the dual benefit of increasing the work in each vector loop iteration and decreasing the overhead introduced by the added strip loop. In this case the PGI compiler unrolls the `do edg` loop regardless of the interchange, and interchanging the k-strip loop results in better optimization in this case. Loop interchange technology is implemented in the PGI compilers, but is not currently enabled in the accelerator optimizer as of PGI 14.4.

4.2 Loop Fusion

After loop interchange, the 3 `do k` loops are adjacent with no intervening code and with identical loop bounds. Fusing these three loops into one larger loop further increases the work per iteration, and eliminates any potential synchronization between loops. The dependence testing to determine legality of loop fusion is more difficult than for loop interchange [Wol96], but in this case it's quite trivial given all array accesses stay in the corresponding `do k` lanes. Loop fusion technology is also implemented in the PGI compilers, but is not enabled in the accelerator optimizer as of PGI 14.4.

```
!$acc loop vector
      do k=1,nvl
         anti_tdcy(k) = 0.
         do edg=1,nprox(ipn)
            if (antiflx(k,edg,ipn) >= 0.) then
               antiflx(k,edg,ipn) =         &
               antiflx(k,edg,ipn)*          &
               min(r_mnus(k,ipn),           &
                   r_plus(k,prox(edg,ipn)))
            else
               antiflx(k,edg,ipn) =         &
               antiflx(k,edg,ipn) *         &
               min(r_plus(k,ipn),           &
                   r_mnus(k,prox(edg,ipn)))
            end if
            anti_tdcy(k) = anti_tdcy(k) +   &
                        antiflx(k,edg,ipn)
         end do
         anti_tdcy(k) =                     &
            -anti_tdcy(k) * rarea(ipn)
         trc_tdcy(k,ipn,nf,t) =             &
            trclo_tdcy(k,ipn,nf,t)          &
          + anti_tdcy(k)
         trcdp(k,ipn,t) =                   &
            trcdp(k,ipn,t)                  &
          + adbash1*trc_tdcy(k,ipn, nf,t)   &
          + adbash2*trc_tdcy(k,ipn, of,t)   &
          + adbash3*trc_tdcy(k,ipn,vof,t)
          tracr(k,ipn,t) =                  &
             max(trmin(k,ipn),              &
                 min(trmax(k,ipn),          &
                     trcdp(k,ipn,t) /       &
                     max(thshld,delp(k,ipn))))
      end do
```

Example 5

Example 5 shows the structure of the new `do k` loop after fusion is performed manually (V2). Loop fusion is typically used to reduce loop overhead, but it also can affect cache behaviour. It can improve performance if the loops operate on the same data and re-use is increased, but can be detrimental if loops become so large and operate on so much data that cache locality is compromised or the register file spills to memory. As we can see in Table 4, loop fusion in this case is beneficial for performance on the GPU.

Version	trcadv1	trcadv2	trcadv3
V0	1240	750	810
V1	1060	610	730
V2	980	540	620

Table 4

As with loop interchange, we expect loop fusion in cases like this will often be profitable on the GPU, because it reduces the number of barrier synchronization points for the vector lanes at the end of vector loops. As we will see, the bigger advantage is enabling other transformations.

4.3 Scalar replacement

Scalar replacement is an optimization whereby an array in a vectorizable loop is replaced with a scalar [CK94, AK02]. This is legal when the k-th element of the array is initialized and used only in the k-th iteration of the vector loop, and is not live out from the loop. In our example, we can perform scalar replacement on the private array variable `anti_tdcy`, reducing memory bandwidth requirements.

```
!$acc parallel num_gangs(10242) &
!$acc vector_length(64) private(anti_tdcy)
!$acc loop gang
        do ipn=ips,ipe
!$acc loop vector
          do k=1,nvl
             anti_tdcy = 0.
             do edg=1,nprox(ipn)
              if (antiflx(k,edg,ipn) >= 0.) then
                 antiflx(k,edg,ipn) =        &
                 antiflx(k,edg,ipn)*         &
                 min(r_mnus(k,ipn),          &
                     r_plus(k,prox(edg,ipn)))
              else
                 antiflx(k,edg,ipn) =        &
                 antiflx(k,edg,ipn) *        &
                 min(r_plus(k,ipn),          &
                     r_mnus(k,prox(edg,ipn)))
              end if
              anti_tdcy = anti_tdcy +        &
                          antiflx(k,edg,ipn)
            end do
            anti_tdcy =                     &
               -anti_tdcy * rarea(ipn)
            trc_tdcy(k,ipn,nf,t) =           &
               trclo_tdcy(k,ipn,nf,t)        &
             + anti_tdcy
            trcdp(k,ipn,t) =                 &
               trcdp(k,ipn,t)                &
             + adbash1*trc_tdcy(k,ipn, nf,t) &
             + adbash2*trc_tdcy(k,ipn, of,t) &
             + adbash3*trc_tdcy(k,ipn,vof,t)
            tracr(k,ipn,t) =                 &
               max(trmin(k,ipn),             &
                   min(trmax(k,ipn),         &
                       trcdp(k,ipn,t) /      &
                       max(thshld,delp(k,ipn))))
          end do
        end do
!$acc end parallel
```

Example 6

Example 6 shows the entire `trcadv3` loop, after loop interchange, loop fusion and scalar replacement have all been performed manually (V3). On an NVIDIA GPU, the compiler may choose to place small gang private arrays such as `anti_tdcy` in shared memory to achieve a similar performance benefit to scalar replacement.

Table 5 shows the performance of the code on an NVIDIA Kepler GPU after each of these successive transformations:

Version	trcadv1	trcadv2	trcadv3
V0	1240	750	810
V1	1060	610	730
V2	980	540	620
V3	980	500	580

Table 5

4.4 OpenACC loop schedules

OpenACC has the concepts of gang and vector parallelism. On an NVIDIA CUDA GPU, gang corresponds roughly to thread-block level parallelism and vector lanes roughly to threads within a thread-block. But that's not a strict definition. An OpenACC implementation can re-map the parallelism more broadly.

In the TRCADV loops, the inner loop has vectors of length 64. The Kepler hardware can support 2,048 active threads in each of the SMX units. With only 64 threads per thread block, it would take 32 thread blocks to completely subscribe a given SMX unit. However, there is a hardware limit of 16 active thread blocks per SMX unit, so we can only half-subscribe the machine if we settle for using the inner loop length of 64 as our vector length. If we take the outer loop and run it across 5121 thread blocks of size 128, dividing each thread block in half to handle two vectors of length 64, it will fully subscribe the hardware.

We can do this using the OpenACC kernels construct using both gang and worker parallelism on the outer loop:

```
!$acc parallel num_gangs(10242) &
!$acc          vector_length(64)
!$acc loop gang
       do ipn=ips,ipe
!$acc loop vector
         do k=1,nvl
```

changes to (V4):

```
!$acc kernels
!$acc loop gang(5121) worker(2)
      do ipn = ips, ipe
!$acc loop vector(64)
         do k = 1, nvl
```

For an NVIDIA Kepler GPU, this improves the performance (because it has improved the occupancy), as reflected in Table 6.

Version	trcadv1	trcadv2	trcadv3
V0	1240	750	810
V1	1060	610	730
V2	980	540	620
V3	980	500	580
V4	1000	430	570

Table 6

Through a sequence of well-understood loop transformations, we improved overall performance of the TRCADV compute kernels by 36%. The OpenACC directives provide the flexibility to specify a better mapping of program parallelism to hardware parallelism, increasing the overall improvement to 40%. The improvement on one of the kernels is nearly 75%. If these transformations can be automated in the compiler, the underlying language and OpenACC directives together will enable a common source formulation that results in performance portability across both SIMD-oriented multi-core CPUs and SIMT-oriented GPUs.

5. Forward performance portability

To enable forward performance portability, we must be able to compile programs written and optimized for today's multi-core CPU architectures to deliver high performance on newer GPU architectures. As we can see from the sequence of steps above, the limitation to forward performance portability in this case is not the hardware, the programming model or the program. It is the inability of the compiler to efficiently map to the GPU hardware a program stylized for multi-core CPUs.

In the case shown above, determining legality of the required transformations is quite trivial. However, as the expressions in the loops become more complicated, legality checks can become a challenge.

Determining profitability is in general a more difficult problem. Compilers use heuristics designed to trigger legal transformations when loops meet basic criteria. Typically these heuristics evolve and improve over time as more and more code examples can be analyzed for a given type of transformation on a given architecture.

The loop interchange described above is now implemented in the PGI compilers internally, and we see about the same performance on the FIM standalone example regardless of how the `do edg` and corresponding `do k` loop are ordered. We are testing this optimization more widely to better understand when it is profitable, and when it must be throttled to avoid unexpected slowdowns before including it as a default optimization in a production release of the compilers.

The loop fusion optimization is somewhat important, mostly because it can reduce memory traffic. We are designing an implementation of loop fusion in the OpenACC code generator, specifically for cases like those seen here. We expect it to be ready for production use late this year.

Scalar replacement has a relatively small impact, and in fact we were somewhat surprised at how small the impact was. The general scalar replacement algorithm described in the literature is quite sophisticated. We expect a much simpler implementation to satisfy all the requirements we see for cases such as shown here. It is likely that our initial implementation will not try to remove the ultimate store to the array, as the cost of detecting when an array is no longer needed is relatively high.

OpenACC does not rigidly define the mapping of program parallelism (gang, worker, vector) to the target machine (grid, block, warp, thread). We are continually experimenting with and incrementally improving the parallelism mapping phase of our OpenACC code generator, called the *Planner*. The Planner must live within the constraints of the target architecture. For instance, on NVIDIA GPUs, synchronization between threads of a warp or all threads in a thread block can be easily implemented, but synchronization between a subset of warps in a thread block is not supported. The compiler must not create a schedule that would require synchronization between execution units (such as warps) that cannot be implemented.

6. Backward performance portability

How have the progressive changes of the original source code from a CPU-friendly formulation to a GPU-friendly formulation affected performance on the CPU? Table 7 shows the results on a single Sandy Bridge CPU core using the PGI 14.4 compiler after each progressive change.

The loop interchange in V1 is not profitable for the CPU, and in fact degrades performance by almost a factor of two overall. This is almost entirely due to limitations in a compiler that only vectorizes innermost loops. In the routines that showed most advantage from SIMD vectorization, the degradation from failure to vectorize (now outer) `do k` loops is almost a factor of 3.

Version	trcadv1	trcadv2	trcadv3
V0	44,500	15,900	60,900
V1	119,400	39,500	69,500
V2	143,000	84,100	75,800
V3	143,800	75,300	68,300
V4	n/a	n/a	n/a

Table 7

Similarly, loop fusion in V2 is not profitable for the CPU. We believe this is due to cache effects discussed earlier, but that has not been verified. Regardless, we have taken another step backward in CPU performance with this transformation. As we progressively optimize for the GPU, we seem to be de-optimizing for the CPU.

The scalar replacement transformation in V3 is profitable for the CPU, presumably because we have reduced memory and SIMD register pressure. We would expect that to be the case generally for both GPUs and CPUs. The V4 transformation has no effect on the CPU code because there is no analog to the gang/vector rescheduling.

The biggest impediment to backward performance portability is the inability of the compiler to vectorize non-innermost loops. Outer loop vectorization has been known technology for a long time. Some compilers have implemented outer loop vectorization by interchanging the outer loop to the innermost position, essentially undoing the V0-V1 transformation. However, there are advantages to vectorizing outer loops without interchanging [DE84, Wol96]. Consider the following loop, a single-precision matrix-vector product added to another vector (smxpy):

```
do j = 1, n2
  do i = 1, n1
    y(i) = y(i) + x(j)*m(i,j)
  enddo
enddo
```

On a computer with a typical vector or SIMD instruction set, the inner loop will perform the following operations:

```
load vector y(i:)
load scalar x(j)
load vector m(i:,j)
multiply x(j) * m(i:,j)
add result to y(i:)
store vector y(i:)
```

The inner loop performs three vector memory operations and only two vector arithmetic instructions, for a compute intensity of 2/3. Instead, what if we can interchange the two loops and vectorize the stride-1 `do i` loop in the outermost position:

```
do i = 1, n1
  do j = 1, n2
    y(i) = y(i) + x(j)*m(i,j)
  enddo
enddo
```

In this form, the overhead for strip-mining the `do i` loop occurs only once, outside the `do j` loop. More importantly, `y(i)` can be accumulated in a vector register in the inner loop; the loading and storing of that register can be moved outside the inner loop. The inner loop then only performs the following operations:

```
load scalar x(j)
load vector m(i:,j)
multiply x(j) * m(i:,j)
add result to y(i:)
```

The compute intensity has improved from 2/3 to 2, with only a single vector memory operation. The ability to keep values in registers across multiple iterations of an inner loop, and even across multiple inner loops, is the strength of outer loop vectorization. When first described, this was called *supervector* performance, and was a motivating factor in the development of BLAS-2.

We believe outer loop vectorization is key to enabling backward performance portability. In fact, we believe outer loop vectorization will be increasingly important for CPUs, as vector and/or SIMD instructions and registers become more critical to performance [NZ08].

Scalar replacement is beneficial for both CPU and GPU because it reduces the memory bandwidth requirements. For both targets, saving a value in a register is always less expensive than saving it to memory, unless the register file becomes a critical resource. Even in that case, spilling the register to memory should not be more costly than storing the value to memory in the first place.

The only question is whether writing the program with fused loops will be too costly for a CPU target. Our experiments show a significant slowdown in all three kernels after loop fusion. We believe that to be caused by poor cache locality. With both the outer `do k` loop and inner `do edg` loop running sequentially, a reference to `antiflx(k,edg,ipn)` will traverse the array down the middle dimension, with a large stride. If the `do k` loop is vectorized, this reference will benefit from sequential access along the stride-1 first dimension. However, this supposition must be proven in practice. An implementation could instead rely on loop fission to optimize memory locality [MCT96].

7. Summary and Conclusions

High performance computer architectures are taking several possible paths. One path will use homogeneous multicore or highly parallel many-core processors at each node, not too different from what we see today. Another path will use a heterogeneous multicore CPU and highly parallel, bandwidth-optimized accelerator at each node. A third path will use a heterogeneous multicore and parallel accelerator integrated on a single chip, perhaps sharing a single path to memory. Likely all three paths, and perhaps others, will coexist for the foreseeable future.

OpenACC is designed to provide expressiveness to the programmer and flexibility to the implementer, and to promote performance portability across a wide range of target architectures. Using a series of program transformations based on existing compiler technology to convert a CPU-optimized representation of an algorithm into one optimized for a GPU, this paper argues that an OpenACC implementation can deliver good performance portability. Moreover, we show how an implementation can transform an accelerator-optimized representation of an algorithm and optimize it for a latency-optimized CPU. We have begun implementation of these methods in the PGI compilers.

About the Authors

Doug Miles is director of PGI compilers & tools at NVIDIA; prior to joining PGI and later NVIDIA, he was an applications engineer at Cray Research Superservers and Floating Point Systems. He can be reached by e-mail at douglas.miles@pgroup.com.

Dave Norton started consulting for PGI over 20 years ago and joined the company full time in 2012. Prior to joining PGI he also consulted for Linux Networx, Liquid Computing, Appro, Quadrics, Microsoft, and other leading HPC companies. Previously he worked for Mission Critical Linux and DEC. He can be reached by email at dave.norton@pgroup.com.

Michael Wolfe joined PGI as a compiler engineer in 1996; he has worked on optimizing and parallel compilers for over 35 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

[AK02] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, 2002.

[CK94] S. Carr and K. Kennedy, *Scalar Replacement in the Presence of Control Flow*, Software-Practice & Experience, 24(1), January 1994.

[CO13] B. Cumming et al, *Refactoring the Community Climate Code COSMO for Hybrid Cray HPC Systems*, CUG2013, Proc. of the Cray User Group Meeting, Napa Valley, Cal., May, 2013.

[DE84] J. Dongarra and S. Eisenstat, *Squeezing the Most out of an Algorithm in CRAY Fortran*, ACM Trans. on Mathematical Software, 10:3, pp. 219-230, Sept. 1984.

[Gov94] M. Govett, *Development and Use of a Fortran to CUDA translator to run a NOAA Global Weather Model on a GPU cluster,* Path to Petascale: Adapting GEO/CHEM/ASTRO Applications for Accelerators and Accelerator Clusters, Urbana, Ill., April 2009.

[Her12] J. Herman et al, *Accelerating Hydrocodes with OpenACC, OpenCL and CUDA*, 2012 SC Companion, Salt Lake City, 465-471, Nov. 2012.

[LL12] J. Larkin and J. Levesque, Application Development for the Cray XK6, Tutorial at CUG2012, Cray User Group, Stuttgart, Germany, April, 2012.

[LSG12] J. Levesque, R. Sankaran and R. Grout, *Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond*, SC'12: Proc. of the Int'l Conf. on High Performance, Salt Lake City, Nov. 2012.

[MFM13] B. T. Minh, M. Forster and U. Maumann, Towards tangent-linear GPU programs using OpenACC, Proc. 4th Symp. on Information and Communication Technology, Da Nang, Vietnam, 27-34, Dec. 2013.

[MCT96] K. McKinley, S. Carr, C.W. Tseng, *Improving Data Locality with Loop Transformations*, ACM Trans. On Programming Languages and Systems, 18(4), pp. 424-453, July 1996.

[NZ08] D. Nuzman and A. Zaks, *Outer-Loop Vectorization – Revisited for Short SIMD Architectures*, Parallel Architectures and Compiler Techniques, Toronto, Canada, pp. 2-11, Oct. 2008.

[Wol96] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley, 1996.