# Performance Portability in Accelerated Parallel Kernels

John A. Stratton, Hee-Seok Kim, Thoman B. Jablin, and Wen-Mei W. Hwu
{stratton, kim868, jablin, w-hwu}@illinois.edu

IMPACT Technical Report
IMPACT-13-01
University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
May 18, 2013
Revision: May 22, 2013

## Abstract

Heterogeneous architectures, by definition, include multiple processing components with very different microarchitectures and execution models. In particular, computing platforms from supercomputers to smartphones can now incorporate both CPU and GPU processors. Disparities between CPU and GPU processor architectures have naturally led to distinct programming models and development patterns for each component. Developers for a specific system decompose their application, assign different parts to different heterogeneous components, and express each part in its assigned component's native model. But without additional effort, that application will not be suitable for another architecture with a different heterogeneous component balance. Developers addressing a variety of platforms must either write multiple implementations for every potential heterogeneous component or fall back to a "safe" CPU implementation, incurring a high development cost or loss of system performance, respectively. The disadvantages of developing for heterogeneous systems are vastly reduced if one source code implementation can be mapped to either a CPU or GPU architecture with high performance.

A convention has emerged from the OpenCL community defining how to write kernels for performance portability among different GPU architectures. This paper demonstrates that OpenCL programs written according to this convention contain enough abstract performance information to enable effective translations to CPU architectures as well. The challenge is that an OpenCL implementation must focus on those programming conventions more than the most natural mapping of the language specification to the target architecture. In particular, prior work implementing OpenCL on CPU platforms neglects the OpenCL kernel's implicit expression of performance properties such as spatial or temporal locality. We outline some concrete transformations that can be applied to an OpenCL kernel to suitably map the abstract performance properties to CPU execution constructs. We show that such transformations result in marked performance improvements over existing CPU OpenCL implementations for GPU-portable OpenCL kernels. Ultimately, we show that the performance of GPU-portable OpenCL kernels, when using our methodology, is comparable to the performance of native multicore CPU programming models such as OpenMP.

## I. Introduction

Heterogeneous computing systems are becoming very prevalent in many market segments. The number of accelerated supercomputers in the Top500 has been steadily increasing since 2009 [1]. The current #1 entry in the Top500 ranking of supercomputers uses an equal number of CPU and GPU chips [1]. Processors marketed towards consumer laptop and workstation systems often integrate heterogeneous CPU and GPU components on the same silicon die. Because GPUs and CPUs developed for somewhat distinct workloads historically, the programming models associated with each are mostly disjoint. Multicore CPU programmers may gravitate towards OpenMP or TBB, while GPU vendors have been encouraging the growth of CUDA [2], OpenCL [3], C++AMP [4], and OpenACC [5].

Divergent programming models and architectures are already causing significant costs in high-performance software development. Optimization is difficult for any architecture, and explicit heterogeneity increases that difficulty even for a single platform. An application developer must target program segments to the appropriate system component, balancing for the relative strengths and weaknesses of each, and then optimize each program segment for the targeted architecture. But software typically outlives systems, and an application with expected longevity has to be targeted towards a system unknown at development time.

When faced with the challenge of targeting an unknown but probably heterogeneous system, developers today typically make one of two choices. Some accept that each performance-sensitive program component must be capable of running on any CPU or GPU with high performance. Those developers bear the high cost of writing high-performance implementations for each, and guarantee performance portability through exhaustive specialization. Other developers lack the motivation or resources to pursue such a high-cost path, and choose instead to target some lowest common denominator, usually the CPU. These two choices drain development effort or leave significant performance opportunities on the table, respectively.

The root problem is a lack of *performance portability*, or the ability to write a single software implementation that can be targeted to either a CPU or GPU with high performance. For some languages, this is because the programming model itself is innately unsuitable for certain architectures. We are unaware of any serious work trying to implement POSIX threads on a GPU, for instance. Other programming models seem to hold much more promise. Stratton et al. presented an implementation of the CUDA language for CPU architectures within two years of the language's release [6]. OpenCL was specifically designed with portability between GPUs and CPUs in mind, with both AMD and Intel releasing x86 CPU implementations. Yet the current ecosystem fails to deliver satisfactory performance portability [7], [8]. This is primarily because a language specification, by itself, is insufficient for establishing performance portability.

Performance portability requires not only an agreed-on functional language specification, but a clear specification or convention for expressing a program's performance-related attributes. In practice, a vendor implementing a language may think first of how the language's functional components might most naturally map to the specific architecture. That implementation will then define some best practices for programmers targeting that architecture through that implementation. The problem is that when the best practices for different platforms diverge, the potential for performance portability is lost. To follow divergent best practices for different platforms, programmers have no choice but to write specialized implementations for each.

OpenCL is a language with both essential characteristics for performance portability in place. The parallelism constructs are abstract enough to transform into diverse lower-level implementations. Additionally, multiple GPU vendors have converged on similar conventions for how OpenCL kernel code should be written for good performance [9], [10], [11].

Our contributions over prior work include:

- Identifying the OpenCL community's emergent performance model.
- Formalizing the patterns necessary for performance portable OpenCL programming.
- Introducing two novel compiler transformations to efficiently map portable OpenCL codes to CPU architectures.
- Providing a best-of-breed OpenCL implementation for CPUs that exploits our performance and portability insights.

We review the OpenCL programming model and discuss the prevailing performance conventions in that model in Section II. While multiple other accelerator languages have similar parallelism models and performance conventions, OpenCL has the most examples of alternative CPU implementation methodologies. These alternative CPU implementations allow us to characterize the impact of failing to match the conventions for expressing performance in one or more significant ways. The characterization and qualitative analysis of current implementation methodologies is presented in Section III.

In Section IV, we describe one possible set of transformations that honors all of the major OpenCL performance conventions while mapping the programming model to a CPU architecture. Section V details the practical implementation of such transformations in an OpenCL compiler and runtime for experimental evaluation. We demonstrate that successfully exploiting the performance conventions achieves greater performance portability than previous work with two primary experiments. First, we show that OpenCL programs following the performance conventions perform as much as 10× better than prior implementations that did not incorporate these performance insights (average 1.8×). Second, we demonstrate that with our implementation, the performance of OpenCL kernels on a

```
1 __kernel void MatMul(__global float *A,
2                 __global float *B, __global float *C) {
3   float result;
4   __global float *A_line = A + get_group_id(1)*A_WIDTH;
5
6   result = 0.0f;
7   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
8
9       for (int ii = 0; ii < TILE_WIDTH; ii++)
10        result += A_line[i + ii] *
11          B[(i+ii)*B_WIDTH + get_global_id(0)];
12
13    barrier(CLK_LOCAL_MEM_FENCE);
14  }
15  C[C_WIDTH * get_group_id(1) + get_global_id(0)] =
16      result;
17 }
```

Fig. 1. A simple, portable matrix multiplication kernel in OpenCL.

CPU is comparable to the performance of a native CPU implementation of the same algorithm in OpenMP. We summarize some additional related work and our conclusions in Sections VII and VIII, respectively.

## II. A Proposed Consensus Performance Model for OpenCL

Clearly, if different vendors have seriously divergent expectations of how programmers should use the language, it is impossible for programmers to write portable code. In the early days of OpenCL, the NVIDIA OpenCL programming guide and AMD GPU programming guide offered conflicting guidelines, mainly regarding whether work-items should operate on scalar or vector data elements [12], [9]. For a variety of reasons, certainly including the pain developers felt from the lack of performance portability, later architectures and compilers from NVIDIA, AMD, and Intel converged on the general conventions described in this section.

The ultimate convention was attractive for several reasons. First, its focus on scalar work-item programs is simpler to use. Tools can aggregate scalar elements into a vector access if necessary, but cannot easily decompose a short vector access into multiple GPU SIMD-lanes [13]. Second, the convention has direct corallaries with dominant performance aspects of modern computer architectures: data-parallel (SIMD) execution, scalable task parallelismsuch, spatial locality, and temporal locality.

Figure 1 shows a listing of an OpenCL kernel for multiplying two matrices in single precision, which we will use as an ongoing example in this paper. If these practices could be performance-portable in theory, then achieving the goal of a portable language will require that both GPU and CPU implementations of OpenCL deliver high performance for codes written with these programming practices.

The kernel exhibits the major performance principles outlined in this section, but also demonstrates a limitation of those principles. The kernel program lacks more advanced algorithm-level data reuse and register tiling techniques commonly used to improve performance on real systems [14]. Nevertheless, the code is sufficient for demonstrating the performance guidelines. Note that these guidelines are primarily about programming for high hardware efficiency, i.e. reaching an achieved bandwidth and execution efficiency close to the architecture's peak. They will not advise whether or not particular algorithms would perform better, or whether bandwidth or execution throughput will be the ultimate limiting performance factor for a given architecture.

### *A. Task and Data Parallelism*

The OpenCL programming model includes a two-level decomposition of work. Although the decomposition is just a two-level hierarchy of parallel tasks, the two levels have very distinct performance implications. All of the work-items in a work-group are guaranteed to be scheduled together, allowing them to coordinate more closely. Work-groups can not make any assumptions about scheduling or co-scheduling of other work groups, which means both that atomic operations must be used to guard critical sections, and in all other ways the groups are constrained to a bulk-synchronous programming model.

The groups of co-scheduled tasks are a clear source of thread-level parallelism, and are exploited in that way by nearly all implementations. Less obviously, perhaps, the work-items within a group are exploited as a source of

vector-level parallelism on all GPU implementations known to the authors. The reasoning is that OpenCL's single-program multiple-data programming model will often naturally lead programmers to write groups of work-items with nearly-identical control flow in many situations. Even if it were not fully natural, the GPU implementations made this programming pattern fastest on their architectures from the beginning. Every known OpenCL GPU programming guide discourages "divergence", or writing programs such that different work-items within a work-group take different paths through the program, making SIMD less effective.

To be performance-portable, the amount of parallelism in and among workgroups needs to be flexible. Work-groups must be at least as wide as the native SIMD width of the machine, but never larger than the machine's capacity to schedule locally and simultaneously. The number of work-groups should be several times larger than the number of processors on the device to enable load-balancing. Given the variety of architectures, it is unclear whether these constraints can be met for all platforms with a fixed-size work-group. At minimum, performance-portable programs have to query the device parameters at runtime and choose group sizes appropriately, or allow the platform itself to choose a group size suitable for itself.

Our example kernel program computes a single output element with each work-item, so the number of work-items for a reasonably large matrix would be substantial. In line 15, `get_group_id(1)`, the second element of the group index tuple, is used as the output row index, indicating that each work-group should process some contiguous section of a particular output row. Therefore, every work-item in a work-group will need to access the same row of data from the $A$ matrix. Also, there are no divergent branches in the kernel. Every condition is independent of the work-item index, so the entire path taken through the program execution is uniform across all work-items. Therefore, SIMD groups of work-items will be fully exercised, without the need to predicate any SIMD lanes at any time.

One interesting point about the OpenCL programming model is that by using groups of work-items as the basis for SIMD execution, the OpenCL implementations are providing the user a way to exploit SIMD without requiring them to program to a specific SIMD width. This is critical for portability, because different CPU and GPU architectures have widely varying SIMD widths. Work-groups that are significantly larger than an architecture's SIMD width enables additional thread- or instruction-level parallelism, as the group can be divided into multiple SIMD-width units. Subsequent proposed languages captured this insight particularly well also, such as the ISPC programming model that advocates SPMD programming as an easy and effective way of writing SIMD code for x86 CPUs [8].

### *B. Spatial and temporal memory locality*

A large part of writing high-performance code is managing data locality well. In a recent survey article of seven broad GPU programming optimization techniques, only one was not directly related to memory locality management [15]. The OpenCL programming model makes explicit certain architectural realities that other languages try to keep abstracted away. Large, coherent memories are inherently more expensive to access than small, local data resources. As typical, we will divide our discussion of locality into two major classes: spatial locality and temporal locality.

Traditional mechanisms for capturing spatial locality were in response to the observation that in sequential programs, if a particular address was accessed, other addresses nearby were likely to be accessed soon in the future. Today, spatial locality is almost a performance requirement, because we build our entire memory systems out of large-line data transactions, such as cache lines and DRAM bursts. Furthermore, building hardware data structures with many ports for independent simultaneous access is very expensive. In particular, this means that even if a wide SIMD unit has gather and scatter capabilities, spatial locality among the addresses accessed will significantly reduce the number of unique memory lines touched by the access, which means a much reduced overall throughput demand on the memory system. In fact, OpenCL programmers are specifically encouraged to assign work-items to data elements such that memory accesses are "coalesced" [9], [10], [11]. Formally, an access is considered coalesced if it can be decomposed into the form:

```
uniform_base_address + (get_local_id(0) % SIMD_WIDTH).
```

A coalesced access causes all of the work-items in a particular SIMD execution bundle to access a set of contiguous elements in memory for the given instruction or expression. To be tolerant to varying SIMD widths

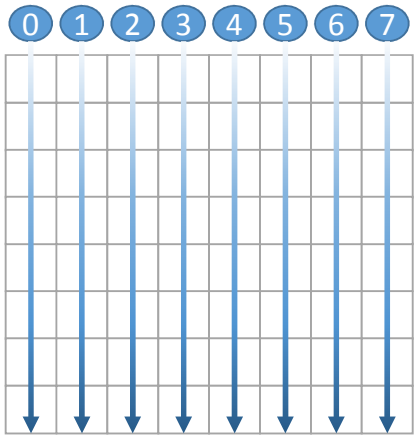


Fig. 2. Access pattern to a tile of matrix $B$ in our example OpenCL program. Tasks within the work group perform wide, coalesced accesses to the memory system.

across architectures, many programs assign the entire work-group to a contiguous set of elements, such that any contiguous subdivision of the group will have a coalesced vector access.

Temporal locality is somewhat more complex to manage portably. Given that inter-group scheduling is out of the programmer's control, her focus is on temporal locality within each group. Futhermore, it is primarily temporal locality in accesses from multiple work-items that needs explicit management; task-private temporal locality is usually handled through simple register promotion. There are two possible approaches to achieving temporal locality in OpenCL: explicitly managed local memory buffers or assuming an implicitly managed cache. Older GPUs prevalent during OpenCL's drafting had very limited caching support, forcing programmers to manage temporal locality through explicitly-managed scratchpads. More recent GPUs from NVIDIA, AMD, and Intel all include memory caches all the way down to the L1 level, which simplifies but does not eliminate the need to specifically consider how temporal locality is managed. Even if a cache is present, it can be exploited one of two ways on a GPU: explicitly controlling the execution order with work-group barriers, or relying on implicit, round-robin scheduling patterns on GPUs to keep all work-items in a work-group roughly in phase with each other. Either of these mechanisms will ensure that memory locations accessed repeatedly by different work-items in the group will likely still be in cache.

In the given example, most accesses to global memory are perfectly coalesced across the entire work-group, because the index expressions on lines 11, and 15 are both of the form `uniform_base + get_local_id(0)`. In the example kernel, the priority of spatial locality is most clearly shown by the access to the input matrix $B$ from line 11, which is shown graphically in Figure 2. The entire work group accesses wide lines of the matrix at a time before proceeding downward in the column direction. Therefore, because each access completely consumes an entire memory line, this kernel achieves a high percentage of peak global memory bandwidth consumption, even though more advanced tiling algorithms could reduce the total number of global memory accesses significantly.

The shown kernel also uses a functionally unnecessary barrier on line 13 to potentially improve temporal locality, particularly for accesses to the matrix $A$. Line 10 shows that the accesses to matrix $A$ do not depend on the local work-item index. Accesses independent of work-item index are uniform across the work-items of the group, which is often an equally effective access pattern for GPUs with any kind of local caching mechanism. The presence of the barrier ensures that a particular tile of the $A$ row remains in cache while all work-items in the group use it.

Note that controlling locality on GPUs always relies on being able to switch between actively executing work-items frequently and with low overhead. Round-robin instruction scheduling is another way of saying that the hardware makes frequent implicit moves between actively executing work-items to balance their progress. Frequent barriers are effectively programmer commands to suspend currently executing work-items at the barrier so that other work-items can catch up. Therefore, we can say that a fundamental requirement of an OpenCL implementation that supports performance portability for current developer practices is a low-overhead mechanism for switching execution between work-items.

```
1 struct wi_state {
2   int local_id[3], group_id[3], global_id[3];
3   float result;
4   float *A_line;
5   int i;
6   int ii;
7   void *restart_point;
8 }
9
10 struct wi_state group_state[WORKGROUP_SIZE];
11 struct wi_state *awi;  //Active work-item
12
13 void barrier(int fence, void* restart) {
14   (awi++)->restart_point = restart;
15   if (awi = group_state + WORKGROUP_SIZE)
16     awi = group_state;
17   goto awi->restart_point;
18 }
19
20 void MatMul(float *A, float *B, float *C,
21             int g_id[3], g_size[3]) {
22   setup_wi_contexts(group_state);
23   awi = group_state;
24 kernel_start:
25   awi->result = 0.0f;
26   awi->A_line = A + awi->group_id[1]*A_WIDTH;
27
28   for (awi->i = 0; awi->i < A_WIDTH;
29        awi->i+= TILE_WIDTH) {
30
31       for (awi->ii = 0; awi->ii < TILE_WIDTH; awi->ii++)
32         awi->result += awi->A_line[awi->i + awi->ii] *
33             B[(awi->i+awi->ii)*B_WIDTH +
34                                    awi->global_id[0]];
35     barrier(CLK_LOCAL_MEM_FENCE, &&restart_0);
36     restart_0:
37   }
38   C[C_WIDTH*awi->group_id[1] + awi->global_id[0]] =
39       awi->result;
40   barrier(0, &&kernel_finish);
41 }
```

Fig. 3. C-like pseudocode representing AMD's OpenCL implementation.

## III. Prior Implementations of OpenCL on CPUs

Much previous work has addressed the challenge of implementing OpenCL on x86 processors, both published academically and implemented industrially. Here, we cover those related works most directly related to our methodology and those that are most popularly used today. We will study each implementation as it relates to the running example in Figure 1.

The AMD CPU OpenCL language implementation is based on the Twin Peaks technology [16]. The primary insight of the implementation is that modern, multicore, superscalar, x86 CPUs support a relatively low level of thread-level parallelism, but a very high degree of instruction-level parallelism. Therefore, it makes most sense to combine all of the work-items in a group into a single CPU thread. The AMD CPU stack accomplishes this with user-level threading techniques, using irregular control flow to "simulate" multiple parallel work-items with a single user thread.

Figure 3 shows a pseudocode example of how this user-level threading is accomplished. First, the implementation declares a data structure suitable for holding all the data private to a single work-item, and then initializes a collection of such data structures to hold the state of all work-items in the group (details not shown.) The CPU thread calls the `MatMul` function with a particular work-group index, which the compiler has modified such that it will complete the execution of all work-items in the specified work-group. It initializes the local state of the work-group on line 22, and selects the work-item with index 0 to be the first *active* work-item. At any given time, the active work-item is the one being advanced through the program. To support multiple work-items with the same kernel code, a level of indirection is added, with `awi` pointing to the private data of the active work-item.

The program execution follows the original OpenCL kernel's operations, referring to the active work-item's private storage through `awi` for references to private variables. Execution of the first work-item continues until the

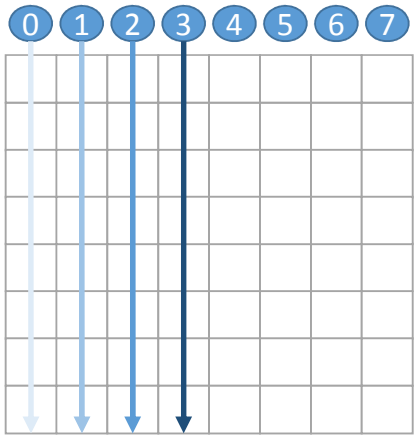


Fig. 4. Access pattern to a tile of matrix $B$ in our example OpenCL program using the Twin Peaks methodology. Only accesses from the first few threads of the work-group are shown.

barrier statement on line 35. At the barrier, the framework saves the program location where the current work-item should be restarted, in this case the label `restart_0` on line 36. It then updates the `awi` pointer to the next work-item's private state, and performs an indirect jump to the location at which the newly activated work-item should be restarted. At initialization, all work-items have their restart points set to the `kernel_start` label, so in this case, the indirect jump takes takes the second work-item to the beginning of the program, as it should. The second work-item will then follow the same program path to the same barrier, at which point the framework will make the third work-item the active work-item, and so on, until all work-items in the group have reached the first barrier.

When the last work-item executes the first barrier, the condition on line 15 will evaluate to true for the first time, and restart the first work-item at `restart_0`, allowing it to continue execution where it left off. It will do so until it reaches the barrier again, where it will again save its current restart point and switch to the second work-item. The constant switching of active work-items continues until work-items begin to reach the kernel's end. At that point, the work-items save their restart point as some sentinel value that will lead the framework to the cleanup code to finish the current work-group, and prepare to execute another work-group if available.

The Twin Peaks methodology has several aspects that make it ill-suited to support the programming practices outlined in Section II. First, the overhead of changing the active work-item is significant. The example shown uses illegal label-passing to illustrate the concepts, but the real implementation is based on `setjmp` and `longjmp`. Even after significant optimization of those low-level routines for this context, the Twin Peaks authors claim an overhead of 10ns or thirty clock cycles per work-item change. Additionally, the micro-threading approach makes no effort to capture vector-level parallelism across work-items. Each work-item is executed in isolation, and any vectorization is limited to opportunities within the code of a single work-item.

Finally, the Twin Peaks implementation does not capture spatial locality as expected by the developer. Figure 4 shows a graphical representation of a single work-group's accesses to the input matrix $B$ over the course of one tile. In a GPU implementation, with wide SIMD vectors and round-robin scheduling, large collections of contiguous addresses are accessed and consumed together. However, a serialization of work-items with the Twin Peaks methodology effectively executes all of a single work-item's accesses first, before the accesses of any other work-items. The kernel follows the guidelines to support SIMD across work-items, leading to interleaved accesses among work-items but strided accesses in the access stream of a single work-item. Figure 4 shows how the serialized implementation accesses a wide range of addresses in a short amount of time for the first work-item, followed by another set of strided accesses from the second work-item, and so on. If the tile size or the memory footprint of the work-group's total state gets large enough, this kind of access pattern will cause significant cache thrashing, and result in very poor spatial locality usage.

Figure 5 shows pseudocode for the result of prior region-based serialization work [17]. Some private variables such as `result` are expanded into an array of values, with one element for each work-item. However, analysis can often detect cases where private variables always store values uniform across the entire work-group, such as

```
1 void MatMul( float *A, float *B, float *C,
2     int g_id[3], // work-group ID
3     int g_size[3]){ // work-group size
4   float result[WORKGROUP_SIZE];
5   float *A_line = A + g_id[1]*A_WIDTH;
6   for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
7     result[__x__] = 0.0f;
8   }
9   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
10
11    for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
12      for (int ii = 0; ii < TILE_WIDTH; ii++)
13        result[__x__] += A_line[i + ii] *
14          B[(i+ii)*B_WIDTH + g_size[0]*g_id[0]+__x__];
15    }
16    //barrier(CLK_LOCAL_MEM_FENCE);
17  }
18  for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
19    C[C_WIDTH*g_id[1] + g_size[0]*g_id[0]+__x__] =
20      result[__x__];
21  }
22 }
```

Fig. 5. C-like pseudocode representing region-based loop serialization.

the variable `A_line`, and avoid creating separate memory locations to store redundant information.

Instead of adding functionality to the barrier function, the compiler uses the very presense of the barrier function to inform analysis of the kernel code. The prior work describes the transformations more rigorously, but effectively, the kernel code is split up into contiguous regions that contain no barriers. Each region is then serialized with an inserted counted loop over the work-item indexes.

In Figure 5, one region occupies lines 6-8, initializing the private variable `result` for all work-items. A second region on lines 11-15 performs the primary computation, accumulating inner products for each column of $B$. The final region on lines 18-21 copies the final results to the correct region of the output space. The code regions themselves constitute nodes in a dynamic control flow graph independent of work-item index, with each dynamic region executed for all work-items. In the example kernel, there is a loop over the second regions, executing it for each tile of the input data, while the first and last regions are executed only once each.

The inserted serialization loops themselves then maintain the semantics of the original barrier, not letting any operations following the barrier in the dynamic execution completing before any operation before that barrier. Therefore, the barrier itself can be removed from the final code, as it adds no information or constraint not already represented by the serialized code.

From a portability standpoint, the region-based serialization methodology has several advantages over the Twin Peaks technique. First, the overhead of executing a barrier is significantly reduced. In this methodology, a barrier only adds a cost of a loop branch and loop counter increment in the worst case. In practice, the overhead is even smaller, because optimizing compilers apply optimizing transformations such as loop unrolling to the serialization loops. Such optimizing loop transformations are practically prohibited by the indirect jumps of the Twin Peaks methodology. Second, this implementation could indirectly result in SIMD vectorization across work-items, if the inserted serialization loops happen to be innermost loops, and a vectorizing compiler is able to conservatively prove the vectorizability of those loops. And finally, the implementation does not fundamentally solve the spatial locality expectation mismatch, as the access patterns remain largely unchanged. The CPU Scalar Access Pattern in Figure 4 still accurately describes the serialized access pattern of the main computation region: strided accesses along a column of the $B$ matrix, followed by more strided accesses along subsequent columns.

Intel's implementation of OpenCL for x86 is both the most recent and the least explicitly disclosed or studied. Our best understanding is that the Intel implementation would behave somewhat like the pseudocode in Figure 6. The figure assumes that the implementation uses region-based serialization for simplicity, but this is not necessarily clear. What is more clear, and noteworthy, is the implementation's focus on explicitly combining multiple work-items into vectorized execution bundles. Instead of creating private, scalar data elements for every work-item, it will create vector data elements for each SIMD bundle, as the declaration of the variable `result` on line 4 shows. All serialization loops are effectively unrolled by a factor of `SIMD_W`, the width of the SIMD units, with each iteration performing operations on vector values.

```
1 void MatMul( float *A, float *B, float *C,
2     // work-group ID and size
3     int g_id[3], int g_size[3]) {
4   simd_float result[WORKGROUP_SIZE/SIMD_W];
5   float *A_line = A + g_id[1]*A_WIDTH;
6   for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
7     simd_store(result[__x__], simd_expand(0.0f));
8   }
9   for (int i = 0; i < A_WIDTH;
10       i+= TILE_WIDTH) {
11
12    for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
13      for (int ii = 0; ii < TILE_WIDTH; ii++)
14        simd_accumulate(&result[__x__] , A_line[i + ii] *
15            simd_load(&B[(i+ii)*B_WIDTH +
16                         g_size[0]*g_id[0]+__x__]));
17    }
18    //barrier(CLK_LOCAL_MEM_FENCE);
19  }
20  for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
21    simd_store(&C[C_WIDTH*g_id[1] +
22        g_size[0]*g_id[0]+__x__], result[__x__]);
23  }
24 }
```

Fig. 6. C-like pseudocode representing Intel's vectorizing OpenCL implementation.

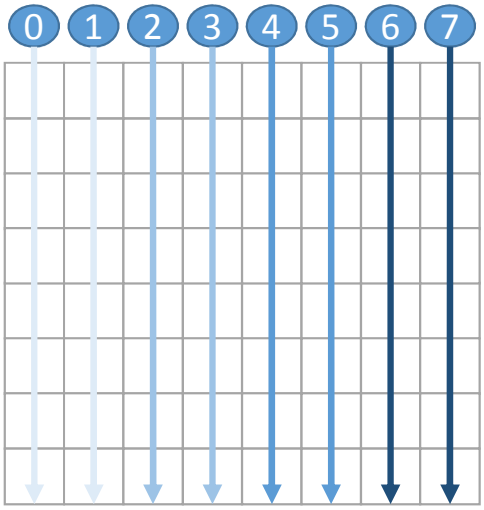


Fig. 7. Access pattern to a tile of matrix $B$ in our example OpenCL program using the work-item SIMD bundling. For illustration, we assume a SIMD width of 2.

In practice, for recent CPUs, Intel's methodology works very well compared to the other techniques already described. It does map multiple work-items to the SIMD units of the architecture, mirroring the expected behavior as described in Section II. The barrier overhead of the implementation is not clear from the disclosed materials, but experimentally seems to be somewhere between the region-based methods and the Twin Peaks method. The explicit combining of work-items into SIMD units does assist in the capturing of spatial locality, but still does not use the caches as effectively as they could. The CPU 2-wide SIMD access pattern in Figure 7 shows why. For GPUs, the effective SIMD width of the processor is very wide, and the cache line size is closely matched to the SIMD data vector width for 32-bit words. In CPUs, while the SIMD widths have increased recently, the cache lines are still significantly larger than the SIMD data vector width. Therefore, a single SIMD access will utilize a smaller portion of the cache line by itself. In a kernel written according to the OpenCL programming guidelines, other work-items in adjacent SIMD bundles would be consuming the rest of that data. However, the overall control flow of the compute region on lines 15-20 of Figure 6 still executes all of the accesses for one SIMD group before any

| | |
|---|---|
| Array Declaration | `int array[ARRAY_SIZE]` |
| Full array slice | `array[:]` |
| Bounded array slice | `array[100:100]` |
| Indirect gather or scatter | `array[indexes[:]]` |

Fig. 8. CEAN array slice notation examples.

accesses from the next SIMD group. The final result is an access pattern that looks like the Figure 7, somewhere between the completely serialized and completely vectorized access patterns.

In summary, in the previous implementation methodologies we show some common performance insights and common portability oversights. It is clear that the work-items in a single work-group should be combined into a single, sequential CPU thread. Work-items within a group are primarily a source of vector- and instruction-level parallelism, both of which CPU architectures exploit from within a single CPU thread. The CPU implementations vary widely in their approach to serializing work-items and capturing SIMD parallelism from the work-items, with the Twin Peaks method vectorizing only explicit vector operations within a work-item, region serialization relying on autovectorization technology, and Intel's methodology directly targeting SIMD instructions. And finally, no current CPU implementation does an excellent job of handling spatial locality given the most common OpenCL programming practices. Instead, they each result int some kind of strided access pattern by executing one or more work-items as long as possible instead of interleaving the accesses of the work-items that would consume the elements of a particular cache line.

## IV. Mapping OpenCL Performance Conventions to CPUs

The previous two sections summarized the implicit performance assumptions held by most OpenCL developers, and how prior work implementing OpenCL on CPUs fails to match some of those assumptions. In this section, we present one potential approach to mapping the performance conventions of OpenCL programs to CPU architectures. The goal of this implementation is to evaluate the practical impact to be had by honoring those performance conventions.

We propose a vector-based serialization of each work-group. Even though the physical SIMD width of a machine is of a fixed and limited value, the programming model's usages would benefit from executing work-groups in a way that emulates a work-group-wide vector machine. If instead of advancing only a small number of work items until they are forced to yield, an implementation could execute each dynamic statement for all work-items in the group before moving on to the next statement. The C Extensions for Array Notation (CEAN) programming model provides an excellent mechanism for describing just such execution semantics.

### A. C Extensions for Array Notation

Intel introduced CEAN as part of their production compiler in 2010. It has also been implemented in gcc, although not integrated into the trunk, and proposed to the C++ standards committee as an industry-standard extension of C and C++ It is very similar to, and likely inspired by, FORTRAN-style array operations. The basic syntax is shown in Figure 8. An *array slice expression* is an array subscript expression (C99 6.5.2.1) that uses an *array slice operator*. The two most relevant array slice operator types are the *full slice* and *bounded slice* operators. A full slice operator, syntactically expressed with a single semicolon as the subscript expression, can only be used on arrays with a known size, and evaluates to the entire contents of the array. A bounded slice operator can be used on any array or pointer, and is a subscript expression of the form:

*array_ptr* [ *base_index* `semicolon` *extent* ].

The base index determines the offset of the first element of the slice, and the extent value determines the number of contiguous elements that should be extracted in the slice. The example bounded array slice in Figure 8 accesses a 100-element slice from the array, beginning with index 100 and ranging to index 199. A bounded slice will always result in an array value with a number of elements equal to the extent. Multi-dimensional slices are permitted, but we will restrict ourselves to single-dimensional array operations for this paper. Array expressions can also be used

```
1 void MatMul( float *A, float *B, float *C,
2     // work-group ID and size
3     int g_id[3], int g_size[3]) {
4   float result[WORKGROUP_SIZE];
5   float *A_line = A + g_id[1]*A_WIDTH;
6
7   result[:] = 0.0f;
8   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
9
10      for (int ii = 0; ii < TILE_WIDTH; ii++)
11        result[:] += A_line[i + ii] *
12          B[(i+ii)*B_WIDTH + g_id[0]*g_size[0]:g_size[0]];
13    //barrier(CLK_LOCAL_MEM_FENCE);
14  }
15  C[C_WIDTH*g_id[1] + g_id[0]*g_size[0]:g_size[0]] =
16    result[:];
17 }
```

Fig. 9. CEAN-based result of our proposed OpenCL implementation.

as array subscript expressions into other arrays, which is useful for defining indirect gather and scatter accesses. Indirect accesses tend to be significantly slower than full or bounded accesses in practice.

Operations on multiple array expressions must operate per-element across the extent of all involved array expressions. For instance, adding a scalar to an array expression will result in a new array expression with the scalar addition applied to every element of the array. Adding a pair of array expressions means an element-wise addition, and requires that the two array expressions have the same number of elements.

### *B. Implementing OpenCL with CEAN*

Figure IV-A shows how we can apply CEAN-style transformations similar to the way previous work applies loop-based serialization. As with previous work, we expand the `result` private variable into an array, because its value depends on work-item index. However, instead of introducing loops over the kernel code, we simply replace the scalar expressions in the code with array slices where appropriate. Accesses to the `result` local variable on lines 7, 11, and 16 use a full slice expression over the array. Accesses to the global memory could use the indirect array access expression syntax in the general case. However, in the example code, all global memory accesses are provably coalesced across the entire work-group. They can therefore be converted into the faster bounded slice operations by decomposing the index operation into the form *base_index* + *get_local_id(0)*. Once the base index expression has been identified, the compiler can generate a bounded array slice beginning at that base index and with an extent equal to the work-group size. For the accesses to `B` and `C`, the compiler must first apply the following equivalence:

```
get_global_id(0) ==
get_group_id(0)*get_group_size[0] + get_local_id(0)
```

The group index and size are then incorporated to the base index expression for the array slice expressions, as seen on lines 12, and 15. The result of all these transformations is a program that expresses the execution of the work-group as a sequence of vector operations over local variables.

CEAN has two important properties that make it very well suited to describing OpenCL work-group execution. First, array expression operations were specifically introduced to support SIMD execution on CPUs. Operations over array expressions are explicitly independent across all elements, and therefore directly targeted as vectorization opportunities. Second, the execution semantics are such that each statement using array slice expressions is evaluated in its entirety before the next statement executes, just as it would if all operations were only scalar. This creates the kind of access pattern that actually achieves the spatial locality the developer intended. And like in the prior region-based serialization approach, barriers are rendered irrelevant in the final code. The array slice ordering constraints essentially provide the same ordering as if there were a barrier between every pair of statements.

Note that this choice of scheduling has strong implications for the local layout of data within the work-group. The Twin Peaks authors specifically defends their choice of storing all the private data for a single work-item contiguously in memory in a data structure. Their claim is that such a layout will get the best spatial locality [16] (although admittedly stating that more research was needed on the topic.) This makes sense given their execute-until-yield serialization model. If one work-item is going to be executed for a long time, it makes most sense that

all its private data would be close together, and not interleaved with the data from other work-items. However, it makes vectorization across work-items inefficient. In order to efficiently combine multiple work-items into a SIMD bundle, all instances of the local variables for work items in that bundle should be contiguously stored.

## V. Practical Implementation and Methodology

To the best of our knowledge, prior implementations of region-based serialization were only available for the CUDA language. For comparisons with a uniform codebase, we implemented both region-based serialization and vector serialization in our compiler, so that each could be compared with each other and with industry implementations.

Our compiler is implemented as a two-stage framework, the first of which performs source-to-source translation from OpenCL to C, performing the high-level transformations in the process. It will transform the kernel code itself into a new function representing the computation of a single work-group. It will also generate a wrapper function to marshall arguments and execute each work-group as an iteration of an OpenMP parallel-for loop. The second phase of the compiler is an off-the-shelf C compiler with OpenMP support, in our case using version 13.0 of Intel's C compiler with the optimization flags `-O3 -xHOST` enabled. The resulting object file is linked with a library implementing all of the OpenCL built-in kernel functions, resulting in a shared library that can be dynamically loaded. The wrapper function for a kernel is invoked by the OpenCL runtime at kernel launch to perform the kernel computation.

In the case of region-based serialization, the high-level transformations performed by the translator include region formation alanysis, as described in previous work [17]. Region formation is the identification of textual regions of input code that can be safely iterated sequentially for all work-items in a work-group. The primary constraints are that regions must be properly nested with the existing control flow constructs within the program itself, and must not contain any barrier operations. Once the regions have been identified, each declared private variable is analyzed for control- or data-dependence on the local work-item index. Those variables that do vary across work-items are expanded into an array of values for each work-item.

When the translator performs vector-based serialization of work-items, it performs the same region formation and work-item-dependence analyses as described above. It then checks the contents of each region to determine whether the region can be serialized with CEAN notation. For instance, loops with work-item-dependent trip counts cannot be expressed in CEAN notation. When such cases are detected, the compiler tries to divide the region into smaller subregions, so that as many statements as possible are within regions that meet the requirements for vector serialization.

Our translator is implemented as an automatic source-to-source transform in the Clang frontend [18] of the LLVM compiler infrastructure [19]. We evaluate our proposed OpenCL implementation on an Intel Core™i7-3770 CPU, with 256-bit vector units supporting the AVX instruction set, using version 13.0 of Intel's C compiler with the optimization flags `-O3 -xHOST` enabled, and running Ubuntu GNU-Linux (Linux kernel version 3.2.0-32).

## VI. Experimental Results

We demonstrate the performance of the proposed approach by running OpenCL and OpenMP benchmarks. We have experimented with two benchmark suites; Parboil [20] and Rodinia [21]. They both come with benchmarks that contains functionally equivalent OpenCL and OpenMP implementations in most cases.

We run the OpenCL benchmarks over four different OpenCL implementations analyzed in this paper. The Intel and AMD implementations are the publicly available versions. The prior region-based serialization proposal and the new CEAN-based vector serialization proposal are both enabled in our own implementation. Figure 10 shows the speedup of the OpenCL implementations relative to the slowest implementation for each benchmark.

The benchmarks that both follow and rely on implicit performance conventions heavily show a dramatic performance increase with our vector-based serialization, including `kmeans`, `sgemm`, `lud`, `cfd`, `backprop`, and `fft`. This is not to say that these benchmarks are simple; the `fft` butterly access pattern is far from trivial to form into coalesced operations, for instance.

Although the benchmarks were written specifically for a GPU architecture, not all of them follow the performance guidelines we describe in this paper. In particular, the `bfs` benchmark is fundamentally a task-parallel algorithm, and does not embody good data-parallelism or memory locality. Other benchmarks may follow the performance

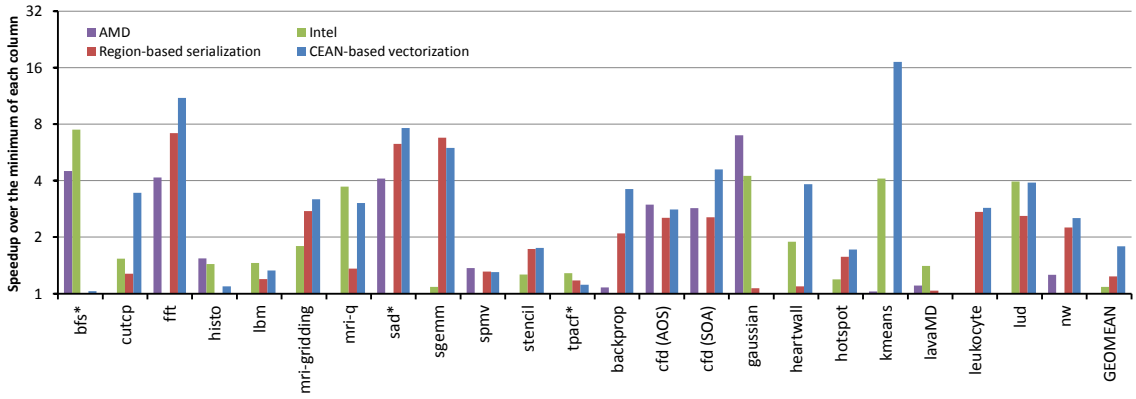


Fig. 10. Comparison of performance of OpenCL implementations. The baseline for each benchmark is an implementation takes the longest execution time.

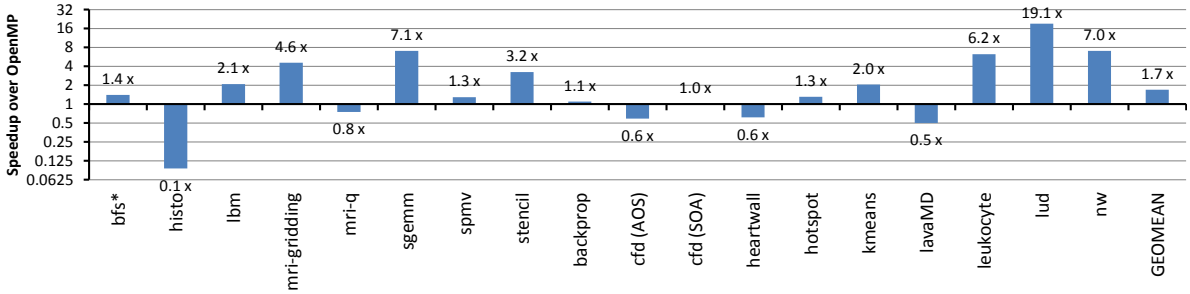


Fig. 11. Comparison of the MxPA OpenCL implementation with OpenMP.

conventions as much as possible, but are naturally less sensitive to implementations that diverge from those performance considerations, such as `lavaMD` or `tpacf`. Such cases usually indicate that the benchmarks use short vector types, enabling the AMD implementation to successfully vectorize operations, and are heavily compute-bound, making them less sensitive to differences in how locality is exploited in the target platform. Still other benchmark have very small runtimes, and we have experimentally determined that the overhead for the OpenMP work distribution used in the region- and vector-based serialization implementations significantly impacts the runtime for the `gaussian`, `bfs`, and `lud` benchmarks. Even when these cases are taken into account, the CEAN vector-based serialization is a full 1.7× faster than AMD's implementation on average.

We also compare the performance of OpenCL kernels on the CPU with OpenMP implementations. Such comparisons are difficult, because the source code implementations can vary significantly. For instance, the default implementation of histo for OpenMP uses a very small degree of parallelism, with significanly less overhead compared to the highly scalable OpenCL implementation. Other cases fall the other way, where the SIMD and locality expressions in the OpenCL kernels were superior to those of the OpenMP implementation, as we can see in lud and sgemm in particular. But with a large number of benchmarks, we can see that, on average, the OpenCL implementations are, if anything, higher-performing on the CPU architecture than the OpenMP implementations, even though the OpenCL implementations were written with a GPU platform specifially in mind. This result boasts the success of performance portability in practice, even if, in theory, the OpenMP implementations could be improved. The OpenCL implementations could also be improved to more closely conform to the OpenCL performance conventions we describe, increasing their performance on the CPU archtecture as well.

## VII. Related Work

Performance portability has been a concern as long developers have had the desire to target multiple architectures. For instance, Jiang et al. were concerned about the performance portability between Multi-socket CPU systems with and without hardware cache coherence as far back as 1997 [22]. Performance portability between different GPU devices was noted a concern by early adopters [23], [24], but that since been tempered by the adoption of the portability guidelines described here, supported by newer tools [13].

Aside from the prior OpenCL and CUDA implementations we discuss at length, other have built systems for executing GPU-style kernels on CPU platforms. Kerr et al. implemented the CUDA programming model on CPUs with the GPUOcelot project [25]. However, performance seemed to be a secondary concern, as the tools is primarily developed to enable better debugging, profiling, and tracing of CUDA applications. It does take a more vector-execution approach, similar to the one we propose, but sacrifices performance for flexibility and the other high-level features mentioned. The Portable Compute Language project [26] aspires to be an open-source, high-performance implementation of OpenCL on CPUs. It has two methods of code generation, roughly corresponding to the region-based and vector-execution methodologies described in this paper. However, the project is somewhat immature, and performance results comparing itself with industry implementations have not been published to the best of our knowledge.

## VIII. Conclusions

Performance portability requires a programming language and well-defined performance convention within that language. The natural pull for a language implementor is to map language constructs to architecture constructs in a straightforward way, minimizing the cost of the implementation. However, implementors for different architectures will be pulled in different directions if this methodology is followed. A collection of implementations guided by architecture idiosyncrasies eliminates performance portability, and creates a huge programming burden for application developers.

However, the community is already moving towards an abstract performance convention for OpenCL for portability between multiple GPU devices. The performance convention embodies preferred methods for expressing data-parallelism, task parallelism, spatial locality, and temporal locality. To achieve performance portability, vendor implementations of the language must determine how to adopt those abstract performance expressions to their architecture, in addition to obeying the functional specification of the language. We have demonstrated that a CPU implementation of OpenCL following those guidelines outperforms the currently available implementations for OpenCL workloads following these conventions.

Programming conventions have limitations. There is some measurable cost paid to conform not only to a language specification but to a particular performance model. The OpenCL benchmarks we studied even included several examples of algorithms that simply have no tenable expression that conforms to the portable performance guidelines. Future work should focus on what the boundaries of the current performance conventions, and add features to the language that broaden the scope of algorithms that can be portably expressed.

Our proposed methodology has been rigorously tested and used for several industry OpenCL application projects. The lead developer of one of those projects, a video processing library, once sent an email with these words. “I just happened to re-run the multi-core performance tests for the VPL today and discovered that the changes we have done recently to improve AMD (GPU) performance have also improved our (CEAN-based) performance.” This is what developers want, the ability to optimize a piece of software once, and have those optimization efforts be favorably reflected on many architectures. We have shown in this paper that performance portability is feasible, when implementations of a language focus on a common performance convention.

## Acknowledgements

This research was conducted with the partial support of the following agencies: the Gigascale Research Center of the Semiconductor Research Corporation, the Microsoft/Intel Universal Parallel Computing Research Center, NVIDIA Corporation, MulticoreWare Inc., and the Department of Energy under the Vancouver project (grant #DE-SC0005515.) The opinions expressed in this paper are solely those of the authors, and do not necessarily reflect those of any organization here mentioned.

# REFERENCES


[1] Top500.org, "Top500 list," Sept. 2012.

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[3] Khronos OpenCL Working Group, "The OpenCL specification, version 1.2," nov 2011.

[4] Microsoft Corporation, "C++ AMP: Language and programming manual, version 1.0," aug 2012.

[5] "The OpenACC application programming interface, version 1.0," nov 2011.

[6] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu, "MCUDA: An effective implementation of CUDA kernels for multi-core CPUs," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pp. 16–30, July 2008.

[7] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS parallel benchmarks in OpenCL," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 137–148, nov 2011.

[8] M. Pharr and W. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *Proceedings of the IEEE conference on Innovative and Parallel Computing*, 2012.

[9] NVIDIA Corporation, "Nvidia cuda programming guide 5.0," 2012.

[10] Advanced Micro Devices, ""amd" accelerated parallel processing "opencl" programming guide," may 2012.

[11] Intel Corporation, "Intel OpenCL optimization guide," Apr. 2012.

[12] AMD, "ATI Stream SDK openCL Programming Guide," Mar. 2010.

[13] M. Delahaye, "Quickly optimize OpenCL applications with SlotMaximizer," 2002.

[14] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 31:1–31:11, IEEE Press, 2008.

[15] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-m. W. Hwu, and N. Obeid, "Algorithm and data optimization techniques for scaling to massively threaded systems," *Computer*, vol. 45, no. 8, pp. 26–32, 2012.

[16] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 205–216, 2010.

[17] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 111–119, ACM, 2010.

[18] "clang: a C language family frontend for llvm."

[19] C. Lattner and V. Adve, ""llvm": A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, (Washington, DC, USA), p. 75, "IEEE" Computer Society, 2004.

[20] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[22] D. Jiang, H. Shan, and J. P. Singh, "Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors," in *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, (New York, NY, USA), pp. 217–229, ACM, 1997.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Comput.*, vol. 38, pp. 391–407, Aug. 2012.

[24] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of opencl programs," in *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[25] A. Kerr, G. Diamos, and S. Yalamanchili, *GPU Computing GEMS Jade Edition, 1st Edition*, ch. 30. Morgan Kaufmann, 2011.

[26] P. Jääskeläinen, C. de La Lama, P. Huerta, and J. Takala, "Opencl-based design methodology for application-specific processors," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 223–230, July.