# PPOpenCL: A Performance-Portable OpenCL Compiler with Host and Kernel Thread Code Fusion


Ying Liu
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
liuying2007@ict.ac.cn

Lei Huang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
leihuang@ict.ac.cn

Mingchuan Wu
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
wumingchuan@ict.ac.cn

Huimin Cui
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Fang Lv
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
flv@ict.ac.cn

Xiaobing Feng
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
fxb@ict.ac.cn

Jingling Xue
School of Computer Science and Engineering, University of New South Wales
Sydney, NSW 2052, Australia
j.xue@unsw.edu.au



## ABSTRACT

OpenCL offers code portability but no performance portability. Given an OpenCL program $X$ specifically written for one platform $P$, existing OpenCL compilers, which usually optimize its host and kernel codes individually, often yield poor performance for another platform $Q$. Instead of obtaining a performance-improved version of $X$ for $Q$ via manual tuning, we aim to achieve this automatically by a source-to-source OpenCL compiler framework, PPOpenCL. By fusing $X$'s host and kernel thread codes (with the operations in different work-items in the same work-group represented explicitly), we are able to apply data flow analyses, and subsequently, performance-enhancing optimizations on a fused control flow graph specifically for platform $Q$. Validation against OpenCL benchmarks shows that PPOpenCL (implemented in Clang 3.9.1) can achieve significantly improved portable performance on seven platforms considered.



## CCS CONCEPTS

• **Software and its engineering → Source code generation**.

## KEYWORDS

Heterogeneous computing, Compiler, OpenCL




*CC '19, February 16–17, 2019, Washington, DC, USA*



ACM ISBN 978-1-4503-6277-1/19/02…$15.00
https://doi.org/10.1145/3302516.3307350

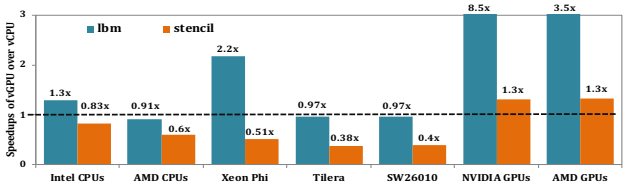
**ACM Reference Format:**
Ying Liu, Lei Huang, Mingchuan Wu, Huimin Cui, Fang Lv, Xiaobing Feng, and Jingling Xue. 2019. PPOpenCL: A Performance-Portable OpenCL Compiler with Host and Kernel Thread Code Fusion. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302516.3307350


## 1 INTRODUCTION

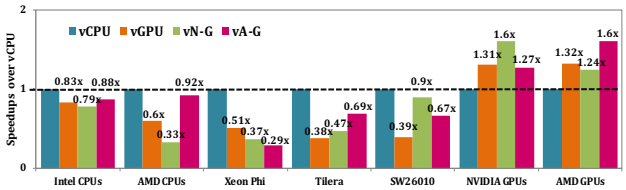
Nowadays, heterogeneous architectures have been extensively adopted in a wide range of computer systems, ranging from mobile devices to supercomputers. Heterogeneous systems are typically equipped with both CPUs and accelerators, such as GPUs. However, the diversity of accelerators makes cross-platform programming a big challenge, thus forcing programmers to write and maintain multiple source code versions for a program on different platforms, e.g., CUDA [28] for NVIDIA GPUs and OpenMP for CPUs.

OpenCL [13] addresses this cross-platform programming challenge by providing a unified parallel programming interface for diverse heterogeneous systems. However, OpenCL guarantees cross-platform portability only in terms of functionality but not performance [8, 32, 34, 39, 54]. Therefore, most OpenCL benchmarks provide multiple source code versions for a program optimized for different platforms, including, e.g., one for CPUs and one for GPUs.

Figure 1(a) shows the performance results of two OpenCL benchmarks, `lbm` and `stencil`, selected from Parboil [46], running on seven platforms (Table 3), with two versions per benchmark, *vCPU* for CPUs and *vGPU* for GPUs. In addition, Figure 1(b) compares the performance results of `stencil` for its four versions, vCPU, vGPU, vN-G (a version written by us for NVIDIA GPUs), and vA-G (a version written by us for AMD GPUs). For a program running on a platform, the speedups of its different versions are given (with vCPU as the baseline, but implicitly in Figure 1(a)).




(a) Speedups of vGPU over vCPU (shown implicitly)




(b) Speedups over vCPU (shown explicitly)

**Figure 1: Performance variations on seven platforms (with the speedups of a program's versions normalized to vCPU).**

Two observations are in order. First, different versions of a program exhibit large performance variations, with no version being the winner across all the platforms. For example, the speedup of vGPU over vCPU for `lbm` is 0.91x on AMD CPUs but 8.5x on NVIDIA GPUs (Figure 1(a)). Second, even for different GPU platforms, sticking to a fixed GPU version for a program does not guarantee the best performance possible for each platform. As shown in Figure 1(b), vN-G (optimized for NVIDIA) is 21.9% faster than vGPU on NVIDIA GPUs but 6.7% slower than vGPU on AMD GPUs. Similarly, vA-G (optimized for AMD) is 21.5% faster than vGPU on AMD GPUs but 3.2% slower than vGPU on NVIDIA GPUs.

Given an OpenCL program specifically written to achieve good performance for one platform, how do we obtain a version of this program that also achieves good performance for another platform? Instead of manual tuning, we aim to achieve this by using a source-to-source OpenCL compiler framework that can automatically apply platform-specific performance-enhancing optimizations to obtain a platform-specific version. Efforts on addressing such cross-platform performance portability issue exist, but mostly for GPUs [23, 53]. Recently, POCL [16] and HPVM [43] are introduced to provide performance-portable OpenCL compilers for different types of accelerators, focusing on optimizing kernel codes only, by applying, e.g., loop optimization, memory optimization, vectorization, and barrier optimization.

However, we have observed that host-code-related optimizations, such as data layout and thread reorganization, are not only platform-specific but also performance-critical. In addition, optimizing the host and kernel codes individually in isolation may not achieve portable performance well. Given an OpenCL program $X$ written for platform $P$ but executed on platform $Q$, we generate a platform-specific version of $X$ for platform $Q$, by (1) fusing $X$'s host and kernel thread codes (with the operations in different work-items in the same work-group represented explicitly), (2) detecting the aliased host and kernel variables (due to the host-device data transfer via kernel arguments) to enable data flow analyses to be applied more precisely on the fused CFG thus obtained, and (3) applying performance-enhancing optimizations, with (1) – (3) geared specifically for platform $Q$.

In summary, this paper makes the following contributions:

- We introduce PPOpenCL, a source-to-source OpenCL compiler for improving performance portability:
  - We propose an approach to build a fused CFG for the host and kernel thread codes of an OpenCL program for a particular platform, WII-CFG (Work-Item Interleaving CFG), which makes explicit the platform-specific execution order for the operations in the work-items of the same work-group.
  - We detect the aliased host and kernel variables via kernel arguments to enable traditional data flow analyses to be applied more precisely on WII-CFG.
  - We describe three platform-specific performance-enhancing optimizations, data layout, thread reorganization and holistic vectorization, on WII-CFG.
- We show the effectiveness of PPOpenCL on achieving portable performance across a variety of platforms.

The rest of the paper is organized as follows. Section 2 motivates our work with an example. Section 3 introduces our compiler framework. Section 4 describes our evaluation. Section 5 discusses the related work. Section 6 concludes.

## 2 MOTIVATION

As a unified parallel programming framework for heterogeneous architectures, OpenCL provides a platform-independent abstract *platform model*, enabling programmers to arrange computations and data references according to the OpenCL *execution model* and *memory model* [13]. In particular, the platform model consists of a host equipped with several OpenCL devices, with each device being divided into several compute units (CUs), which are further divided into several processing elements (PEs). The memory model defines two memory regions, the host memory and the device memory, which are available to the host and kernel codes, respectively. The execution model is defined in terms of two distinct units of execution, the kernel code running on several OpenCL devices and the host code on the host. When a kernel is submitted for execution, an index space, i.e., NDRange is defined, in which each point is a work-item (kernel thread) running on one PE. The work-items are organized into work-groups, with each work-group running on a CU.

To support the OpenCL programming model, an OpenCL compiler framework usually consists of two compilers, the host compiler and the kernel compiler, for a platform. When compiling an OpenCL program, the host compiler compiles the host code and links it with the OpenCL libraries. The kernel compiler, invoked when the program execution starts, generates the executable code for one work-item via the *clBuildProgram()* API. Then the executable code is duplicated for all work-items in the same work-group, dispatched to different PEs via the *clEnqueueNDRangeKernel()* API.

This dichotomy of host and kernel compilers aims largely to achieve code portability across different platforms. However, performance portability cannot be guaranteed.

**Host code** (with lines in yellow as calls to the OpenCL library)

```
1  main(...) {
2    //neighbors of atoms declared and initialized
3    int h_n[nA*nN]; float h_f[nA],h_p[nA]; cl_mem d_n, d_f, d_p;
4    for (i =0;i<nA;i++)
5      for(j=0;j<nN;j++)
6        h_n[i+j*nA] = neighborIter[i][j];
7    clCreateBuffer(d_n, nA*nN*sizeof(int)); clCreateBuffer(d_f,...);clCreateBuffer(d_p, ...);
8    clEnqueueWriteBuffer(d_n, h_n, nA*nN*sizeof(int)); clEnqueueWriteBuffer(d_p, h_p, ...);
9    //kernel compiler invoked to generated code for one work-item
10   clCreateProgramWithSource(prog,"kernel.cl"); clBuildProgram(prog);
11   //task granularity determined
12   clSetKernelArg(ker, 0, d_f); clSetKernelArg(ker,1,d_p);
13   clSetKernelArg(ker, 2, d_n); clSetKernelArg(ker,3,nN);
14   clSetKernelArg(ker, 4, nA); ...
15   g_size=[nA,1,1];l_size=[128,1,1];
16   //work-items dispatched for execution
17   clEnqueueNDRangeKernel(ker, g_size,l_size);
18   clFinish();
19   //results transferred back from devices to host
20   clEnqueueReadBuffer(h_f, d_f, nA*sizeof(float));
21   //post-processing on the results received
22   compute(...);
23 }
```

**Kernel code**

```
1  __kernel ker(__global float* f,
2               __global float* p,
3               __global int* n,
4               int N, int A...) {
5    //computation for one work-item
6    tid=get_global_id(0);
7    for (j=0; j < N; j++) {
8      idx=n[tid+j*A]; pos=p[idx];...
9    }
10   f[tid]=...
11 }
```

```
...
  for (i=0;i<nA;i++)
    for(j=0;j<nN;j++)
      h_n[i*nN+j]=neighborIter[i][j];
...
```

```
...
    for (j=0; j < N; i++) {
      idx=n[tid*N+j]; pos=p[idx];...
    }
...
```

**Figure 2: An OpenCL program for md.**

Figure 2 illustrates the challenge faced in achieving performance portability with an OpenCL program abstracted from md, in the SHOC benchmark suite [6], initially written for GPUs. This program performs an *nbody* simulation of *nA* atoms based on its *nN* neighbors. The host code consists of lines 3-6 (for declaring the atoms and their neighbors used) and line 22 (for post processing the results returned from the kernel code) in green, as well as lines 7-20 (for calling appropriate APIs in the openCL library) in yellow. The kernel code consists of lines 1-11 (for computing one atom's result based on its neighbors in one work-item) in red.

When md is compiled for CPUs, the kernel compiler will find that the data layout of **n** is not cache-friendly, but no avail since this is dictated by lines 4-6 in the host code. Therefore, programmers are expected to write another better-performing version for CPUs. To automate this process, PPOpenCL will fuse the host and kernel thread codes together and transform both simultaneously, with the modifications in blue. The modified version will be compiled by the host and kernel compilers to run more efficiently on CPUs.

## 3 THE PPOPENCL FRAMEWORK

Figure 3 depicts the flow chart of PPOpenCL. Given an OpenCL program, PPOpenCL will turn it into a platform-specific OpenCL version in three phases: (1) *control flow analysis* for building a fused CFG for its host and kernel thread codes, (2) *data flow analysis* for capturing the data flow across the fused CFG, and (3) performance-enhancing optimizations for improving the performance of the platform-specific version generated on the fused CFG.

Currently, all the analyses are static, without relying on any profiling information related to host and kernel codes.

### 3.1 Control-Flow Analysis

The objective is to build a CFG for the fused host and kernel thread codes of an OpenCL program on a given platform. This is done in two steps. First, we inline the CFG of a kernel inside the host program, obtaining an *inlined CFG* (Section 3.1.1). Second, we replicate the CFG of a kernel by representing explicitly the execution order for the operations in the work-items of the same work-group, thereby enabling our later data flow analyses and optimizations.

*3.1.1 Inlining Kernels Inside the Host Program.* We traverse the CFG of the host code (*host CFG*), built by the host compiler, looking for a kernel launching API invocation. When one is found, we inline its CFG (*kernel CFG*), built by the kernel compiler, in the host CFG. As is standard, a call (return) edge from this call site (the exit of the kernel CFG) to the entry of the kernel CFG (this call site) is

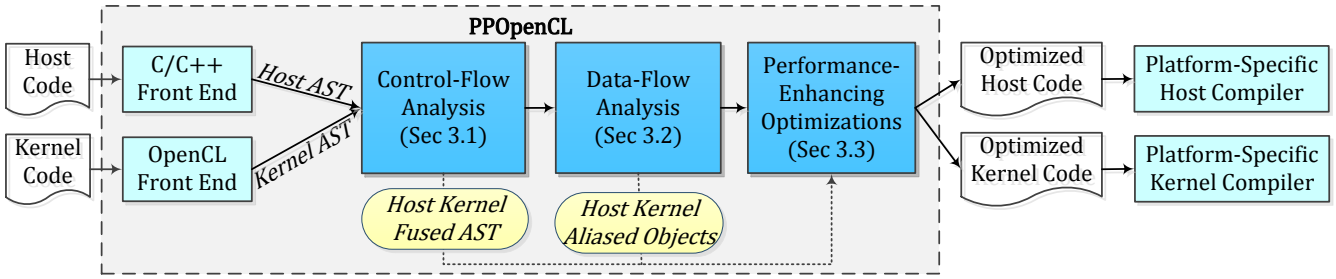



**Figure 3: The PPOpenCL compiler framework.**

also added. This inlining process works for multiple kernel launching points, possibly inside loops with arbitrary control flow.

At this stage, the CFG obtained is called an *inlined CFG*.

*3.1.2 Modeling the Execution Order of Work-Items.* For the OpenCL execution model, it suffices to model the execution order of the operations in the work-items from the same work-group. Their execution order is platform-dependent. A platform can choose to implement the OpenCL execution model in different ways, thereby introducing different execution order constraints. For example, the work-items in a work-group are executed in the SIMT manner on NVIDIA GPUs and serially on AMD CPUs and Tilera.

*WII Functions.* To express such execution order, we associate a function, a *WII (Work-Item Interleaving) function*, with the call edge of each kernel (in the inlined CFG). For a given work-item, all its operations are executed serially in a sequence and can thus be identified by an integer (starting from 0), representing its position in the sequence. The operations inside branches are guarded. For an operation *op* in a kernel's work-item identified by is thread id (*local_id*) *tid* (starting from 0) for platform $P$, $WII^P(tid, op)$ is an integer that specifies the (logic) step at which *op* is executed.

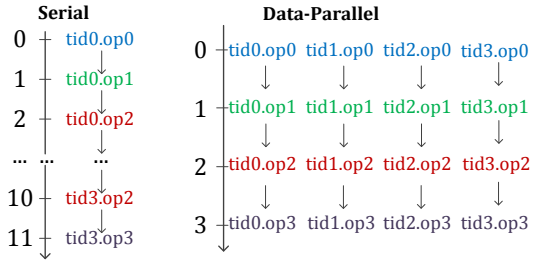



**Figure 4: Two representative execution orders.**

Figure 4 depicts two representative execution modes, "Serial" and "Data-Parallel". In each case, the right column lists the operations, indexed by *tid.op*, in a work-item, and the left column gives their execution order. Table 1 gives their corresponding WII functions, together with some representative platforms supporting these execution orders. All these seven platforms are taken from Table 3.

*Obtaining the Fused CFG.* Given the inlined CFG obtained, we can build the fused CFG, called *WII-CFG*, by making explicit the execution order of the operations in the work-items in a work-group for a kernel according to its WII function. This is a simple process of replicating the kernel CFG by grouping together the operations in different work-items scheduled together and adding the control flows where appropriate across these groups. A group of operations sharing the same opcode is identified as a vector operation. In the case of multiple kernels running on different architectures, different target-specific WII-functions will be used. Note that execution-order-sensitive built-ins, such as barriers, are handled according to [16, 22], and execution-order-insensitive built-ins, such as atomics, don't affect correctness. Currently, some advanced built-ins, such as device-side `enqueue_kernel()`, are not yet optimized.

Figure 5 shows how to build the WII-CFGs using the WII functions in Table 1 for a kernel as shown. For serial execution, the kernel CFG is replicated conceptually rather than physically as many times as the number of work-items, $M$. For data-parallel execution, the number of replications is $\lceil M/N \rceil$ times (where $N = 2$ is the degree of parallelism), with the identically numbered operations from $N$ work-items replaced by one vector operation.

## 3.2 Data Flow Analysis

In the OpenCL memory model, the host and device have separate memory spaces. Thus, programmers need to explicitly manage the data transfer between the two.

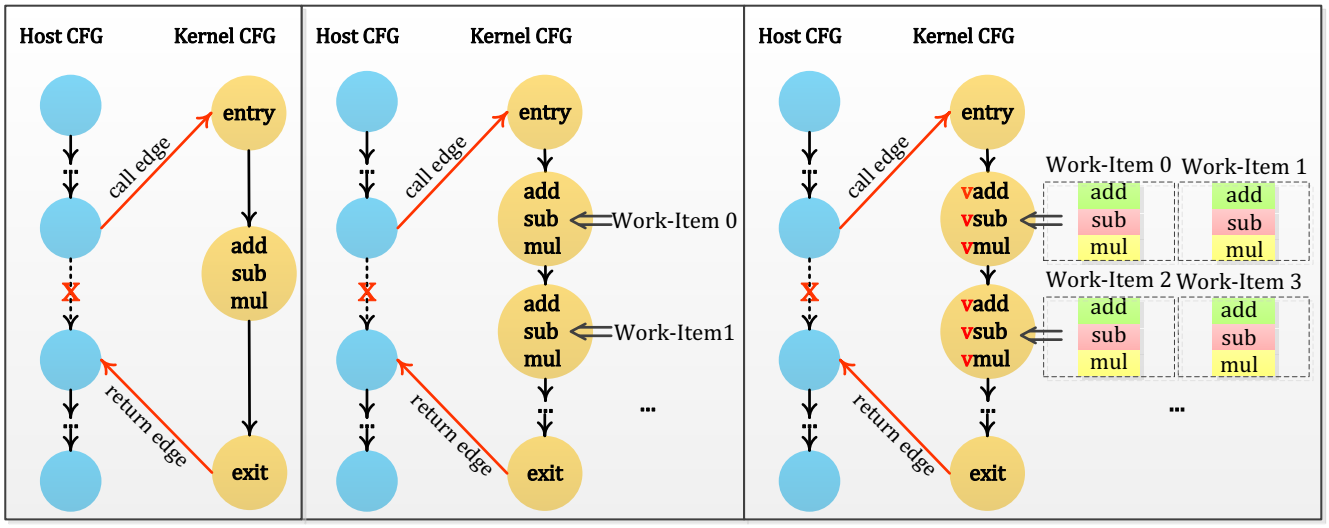
For a shared memory space, two variables are aliases if they point to the same data. This notion carries over naturally to separate memory spaces. In this paper, a host object (residing in the host) and a kernel object (residing in the device memory) are said to be *aliased* if both operate on the same data via a kernel argument (made possible due to the explicit host-device data transfer between the objects).

In Section 3.2.1, we discuss how to identify such aliases. In Section 3.2.2, we describe briefly how traditional data flow analyses can be then applied (more precisely).

*3.2.1 Identifying Aliased Host and Kernel Objects.* The aliased host and kernel objects are found simply by traversing the host program. For buffers (one of the most widely used data structures), these aliases can be found by looking for calls to `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` (indicating the direction of the data transfer), as well as `setKernelArg()` (for the corresponding kernel parameter connecting the aliased objects).

**Table 1: The WII functions for the two execution orders depicted in Figure 4, supported by some platforms. For "Data-Parallel", $N$ is the degree of parallelism, which is 4 on Intel Xeon CPUs, 16 on Xeon Phi, 32 on NVIDIA GPUs, and 64 on AMD GPUs.**

| Execution Mode | WII Function | Representative Platforms |
|---|---|---|
| Serial | $num\_of\_op * tid + op$ | AMD CPUs/Tilera/SW26010 |
| Data-Parallel | $num\_of\_op * \boxed{\frac{tid}{N}} + op$ | Intel CPUs/Xeon Phi/NVIDIA GPUs/AMD GPUs |




(a) Inlined CFG (b) Serial CFG (c) Data-parallel CFG

**Figure 5: Constructing the WII-CFGs in (b) and (c) using the two WII functions in Table 1 from the inlined CFG in (a). In (c) the degree of parallelism is assumed to be $N = 2$.**

Table 2 gives the three aliases for the buffers in Figure 2. For example, the host object **h_n** and the kernel object **d_n** are aliased via `clEnqueueWriteBuffer()` in line 8, where **d_n** is specified as the 2nd argument of a call to "ker" in line 13.

The aliases for other types of objects can be detected similarly. Take images for example. The API calls for establishing their aliases are `clEnqueueWriteImage()` and `clEnqueueReadImage()` instead.

A scalar in the host program may be indirectly assigned to a formal parameter of a kernel also using `clSetKernelArg()`. For example, in Figure 2, $nN$ ($nA$) in the host program is assigned to $N$ ($A$) in line 13 (line 14). In essence, their underlying objects are actually aliased and also detected.

**Table 2: The aliases host and kernel objects in Figure 2.**

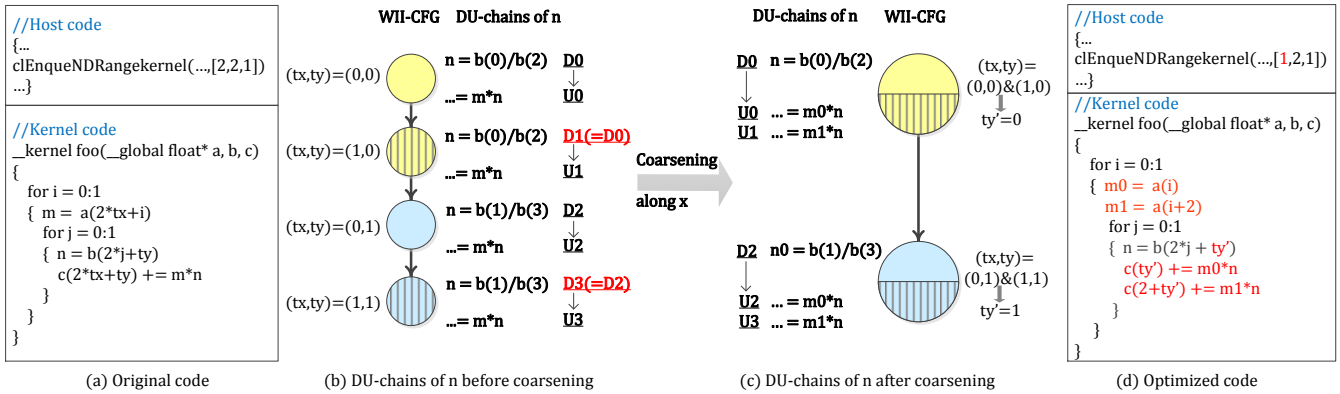
| Host Object | Kernel Object | Kernel Argument | Transfer Direction |
|---|---|---|---|
| h_f | d_f | ker (0th) | read |
| h_p | d_p | ker (1st) | write |
| h_n | d_n | ker (2nd) | write |

*3.2.2 Applying Traditional Data Flow Analyses.* Traditional data flow analyses, such as du-chain and liveness analyses, can now be applied to WII-CFG in the usual manner, except that the aliases detected above are used.

Let us revisit the program in Figure 2 that is originally written for GPUs but now intended to run on CPUs. The kernel compiler, which compiles "ker" alone, cannot improve its poor cache locality due to the strided accesses to **n**. In PPOpenCL, however, **n** is known to be aliased with **h_n** (Table 2), as they operate on the same data. Therefore, the locality-enhancing transformations as shown in blue are performed automatically. In more complex cases, some data flow analysis is needed, assisted by such alias information.

### 3.3 Performance-Enhancing Optimizations

Given the new optimization opportunities exposed across the boundaries of the host and kernel thread codes for a platform, PPOpenCL will now apply performance-enhancing optimizations to the fused CFG to produce a platform-specific OpenCL program. This optimized program will be compiled by the host and kernel compilers to run on the platform. Note that PPOpenCL focuses on optimizations requiring both host and kernel transformations, host-only or kernel-only optimizations, such as adjusting work sizes and optimizing local memory usage, can be applied to the optimized codes produced by PPOpenCL, to obtain further performance gains.




(a) Original code (b) DU-chains of n before coarsening (c) DU-chains of n after coarsening (d) Optimized code

**Figure 6: Thread reorganization via coarsening.**

Below, we describe three important new optimizations, (1) thread reorganization, (2) data layout optimization, and (3) holistic vectorization, that we have added in PPOpenCL, all focusing on the work-items in the same work-group. They are applied in the order given, i.e., (1) – (3), as (1) exposes opportunities to (2) and also forms the basis for (3). Furthermore, (2) can also open up new opportunities for (3).

*3.3.1 Thread Reorganization.* The OpenCL program model encourages a large number of work-items, i.e., threads to be created in a work-group in order to balance workload and hide latency during runtime thread scheduling, at the expense of introducing redundant operations across the work-items. Such redundancy can result in runtime overhead for some platforms such as CPUs equipped with effective thread schedulers. Therefore, a coarser granularity can be obtained with thread coarsening.

We achieve this by performing a thread reorganization that applies a coarsening factor to a particular dimension in the NDRange of a work-group. In particular, PPOpenCL examines the inter-thread du-chains in WII-CFG and picks the best dimension to coarsen in order to maximize the amount of redundancies removed, without significantly reducing the amount of parallelism exploited in the original program.

Figure 6 gives an example with four work-items in a work-group. For the code in Figure 6(a), its du-chains for $n$ are given in Figure 6(b). By coarsening the $x$ dimension by a factor of 2, work-items $(0,0)$ and $(1,0)$ are merged and work-items $(0,1)$ and $(1,1)$ are also merged. As a result, some redundant operations in each pair of two merged work-items have been eliminated, as revealed in Figure 6(c). Finally, the coarsened code is given in Figure 6(d). For this example, the $y$ dimension can be coarsened in a similar way.

The thread reorganization optimization is done according to Algorithm 1. We find the best NDRange dimension $d$ to coarsen with a factor of $f_c$ by maximizing the execution time reduced (line 16), computed from the redundant operations eliminated (line 9), subject to some constraints on the amount of parallelism lost (line 15, i.e., Algorithm 2). Finally, host and kernel codes are transformed accordingly (lines 18 – 19).

**Algorithm 1** Thread Reorganization

1: $Cycle_{red} = [0,0,0]$ // cycles saved for dimensions $x$, $y$, $z$
2: $F_c = [0,0,0]$ // coarsening factors for $x$, $y$ and $z$
3: **for all** variables $v$ **do**
4: $du\text{-}chain \leftarrow WIICFG.Kernel().BuildDU(v)$
5: **for all** dimensions $d$ in $\{'x','y','z'\}$ **do**
6: **if** $Coarsenable(du\text{-}chain, d)$ **then**
7: $F_c[d] = local\_size(d)$ // size of dimension $d$
8: **for all** reducible operations $op$ of $v$ **do**
9: $Cycle_{red}[d] += op.latency$
10: **end for**
11: **end if**
12: **end for**
13: **end for**
14: **if** $F_c \neq [0,0,0]$ **then**
15: $F_c = ApplyConstraints(F_c, \{'x','y','z'\})$
16: Let $d$ be the dimension such that $Cycle_{red}[d] * F_c[d]$ is the largest (among dimensions $x$, $y$ and $z$)
17: $f_c = F_c[d]$
18: $CoarsenThreads(WIICFG.Kernel(), d, f_c)$
19: $ChangeNDRange(WIICFG.Host(), d, f_c)$
20: **end if**

**Algorithm 2** ApplyConstraints($Factor$, $Dims$)

1: **if** $WIICFG.type() == "GPU"$ **then**
2: **for all** dimensions $d$ in $Dims$ **do**
3: Let $s$ be the size of dimension $d$ of a work-group and $s_1 \times s_2$ the sizes of its other two dimensions
4: Adjust $Factor[d]$ so that it is still the largest satisfying (1) $s/Factor[d] \times s_1 \times s_2 \geqslant 128$ and (2) $32 \mid (divides)\ s/Factor[d] \times s_1 \times s_2$
5: **end for**
6: **else if** $WIICFG.type() == "Intel"$ && $'x' \in Dims$ **then**
7: $Factor['x'] = 1$
8: **end if**
9: **return** $Factor$

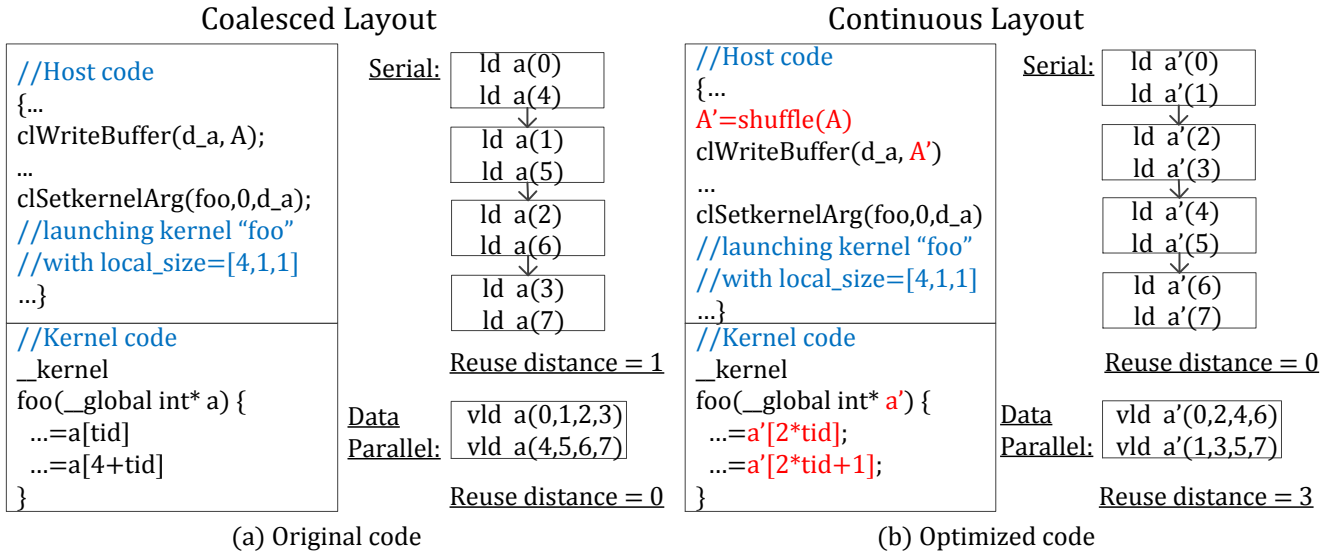



**Figure 8: Data layout optimization for an example (with two int elements per cache line and four work-items per work-group).**

In `ApplyConstraints()`, constraints are introduced to reduce the amount of parallelism lost. For GPUs, we require the two conditions stated in line 4 to hold for every coarsened dimension (as recommended in [28]). For data parallel platforms such as Xeon Phi and Intel CPUs, the coalescing factor for the $x$ dimension is simply 1 (line 7) since the work-items in a work-group will be vectorized along this dimension (implying that this dimension will not be coarsened). For any non-divisible coarsening factor, global/local work-sizes are simply rounded up (with non-ops inserted) [41].

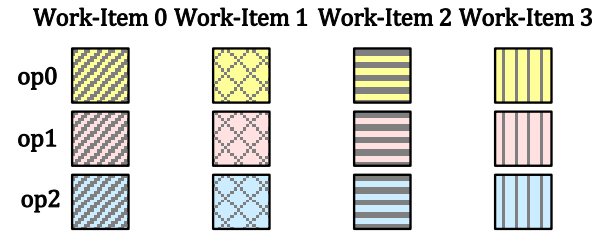
The amount of control divergence may reduce the amount of inter-work-item redundancies that can be potentially eliminated by the kernel compiler. In the absence of control divergence, the exposed redundancies can be eliminated with CSE (Common Subexpression Elimination). In the presence of control divergence, PRE (Partial Redundancy Elimination) [52] may be used instead.

*3.3.2 Data Layout Optimization.* As described in Section 2, a platform dictates how the work-items in a work-group are executed. When a buffer is accessed by multiple work-items, our optimization selects one of the two data layouts, as illustrated in Figure 7, and performs the required code transformation for the platform.

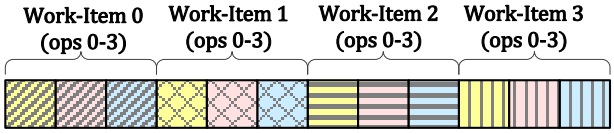
For GPUs, a coalesced layout is always preferred as this option enables global memory coalescing to be performed.

For the other data-parallel platforms (excluding GPUs) and serial platforms (with a few representatives listed in Table 1), PPOpenCL picks a data layout for the purposes of improving the cache locality of their on-chip memory, based on the (cache line) reuse distance metric [7]. For platforms such as SW26010 [11] with a compiler-managed scratchpad memory, the cache line size is assumed to be the scratchpad size.

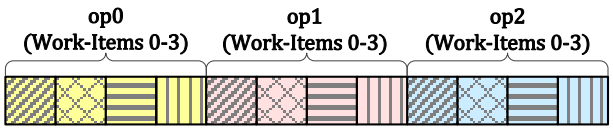
The objective of this optimization is to select a suitable data layout for a kernel object (e.g., a buffer) assessed in a kernel. Note that the elements in such a buffer can be transferred to the local memory in the same way as they are transferred to registers. Hence, no complication should arise when the local memory is involved. For a buffer (accessed in a kernel), its reuse distance, which is computed on WII-CFG, is platform-dependent. Given a platform, PPOpenCL selects one of the two data layouts (shown in Figure 7) with the




**(a) Different work-items access different buffer elements**




**(b) Continuous layout with intra-work-item locality**




**(c) Coalesced layout with inter-work-item locality**

**Figure 7: Two data layouts for a buffer.**

**Algorithm 3** Data Layout Optimization

```
1:  for all kernel objects A do
2:      assert(A's layout is continuous or coalesced)
3:      curRD = CalculateRD(WIICFG.Kernel(), A)
4:      if curRD == 0 then
5:          continue
6:      end if
7:      tmpKernel ← OtherLayout(WIICFG.Kernel(), A)
8:      otherRD = CalculateRD(tmpKernel, A)
9:      if otherRD < curRD then
10:         WIICFG.SetKernel(tmpKernel)
11:         Shuffle(WIICFG.Host(), AliasedObj.Get(A))
12:     end if
13: end for
```

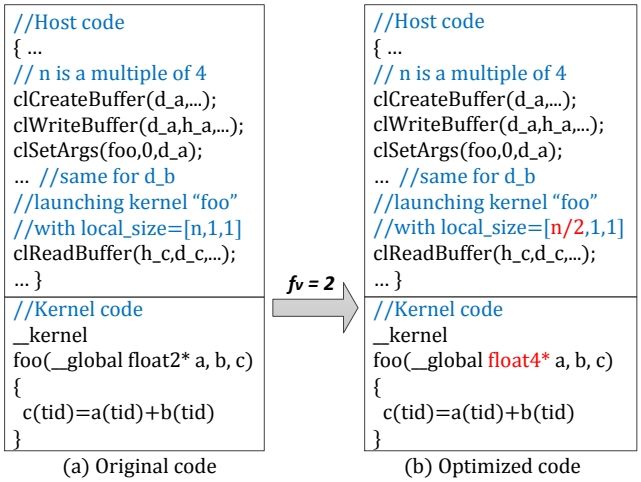
smaller reuse distance according to Algorithm 3. PPOpenCL applies this optimization only to an object with a continuous or coalesced layout (line 2). The reuse distances for the original and alternate layouts are calculated (lines 3 and 8), based on a standard locality analysis algorithm [26], by grouping array references (e.g., buffers as 1-D arrays) in loops (created after the CFG fusion) in terms of their array indices. Finally, the layout with the smaller reuse distance is selected (and transformed if necessary).

Currently, the data layout optimization is conservative. Given a buffer represented by a variable $v$, this optimization is applied only when it is accessed via $v$ itself rather than also its aliases (found based on its du-chains and ud-chains). A more sophisticated alias analysis [47] may be needed to expose more opportunities for the optimization.

Figure 8 illustrates our optimization for a small OpenCL program, assuming two elements per cache line (for the on-chip memory) and four work-items per work-group. In the original program given in Figure 8(a), which adopts a coalesced layout for buffer `a`, its reuse distance is 0 for a data-parallel platform and 1 for a serial platform. Therefore, PPOpenCL will retain this layout for a data-parallel platform. However, for a serial platform, PPOpenCL will produce the optimized code given in Figure 8(b), which adopts a continuous layout for buffer `a'` instead. Its reuse distance is now 3 for a data-parallel platform but 0 for a serial platform.

Note the presence of the `shuffle()` function provided by PPOpenCL in the optimized code, which is inserted by line 11 of algorithm 3. Based on the du-chain for `A`, `A' = shuffle(A)` is inserted before the kernel call. Similarly, a corresponding `shuffle()` call that restores the original layout of `A` after the kernel call is inserted (if needed).

*3.3.3 Holistic Vectorization.* The work-items in a work-group can be vectorized in three different ways. First, inter-work-item vectorization (i.e., loop level vectorization [56–58]), as shown in Figure 5(c), is supported by platforms such as Xeon Phi and Intel CPUs. Second, intra-work-item vectorization (i.e., Superword level parallelism [21, 56]), can be done by the kernel compilers on the platforms equipped with SIMD engines. Finally, programmers can explicitly vectorize the operations in a work-item by using vector types (e.g., float2 and float4) or the attribute `vec_type_hint(<type>)`.



(a) Original code (b) Optimized code

**Figure 9: Holistic vectorization (assuming a vector length of 4 floats). `d_a` can be padded if $n$ is not divisible by 4.**

These three decisions, inter-work-item vectorization (Inter-Vec), intra-work-item vectorization (Intra-Vec) and explicit vectorization (Exp-Vec) often yield different performance benefits.

Inter-Vec is powerful for OpenCL programs due to the lack of the dependencies across different work-items. However, it is only supported by platforms such as Intel CPUs and Xeon Phi. We have implemented this technique in PPOpenCL for other platforms. Given a platform, our holistic vectorization attempts to apply one of the three techniques, Inter-Vec, Intra-Vec and Exp-Vec, or combine some of these in order to improve the overall performance benefit obtained.

This optimization is applicable to both CPUs (equipped with SIMD engines) and GPUs (supporting vector loads/stores, e.g., 64-bit and 128-bit loads and stores on NVIDIA GPUs).

As shown in Algorithm 4, PPOpenCL first attempts to apply Inter-Vec to a work-group, when vector loads and stores are possible (to avoid more expensive gather/scatter operations). When vectorizing a work-group, PPOpenCL maximizes the number of work-items, $f_v$, by maximizing the number of cycles reduced (lines 1 – 2), with $f_v$ satisfying the same parallelism-inhibiting constraint introduced by `ApplyConstraints()` but for dimension $x$ only (line 3). In addition, one additional constraint on register pressure is imposed, $f_v \leqslant \frac{reg_{\text{all}}}{reg_{\text{wi}}}$, where $reg_{\text{all}}$ is the maximum number of registers allocated to a work-item (an architecture-specific constant) and $reg_{\text{wi}}$ is simply the number of variables available (to approximate the number of registers used), in order to avoid introducing register spills (lines 4 – 5).

If Exp-Vec is present, its explicitly vectorized operations are considered by Inter-Vec (lines 6 – 7). If both Exp-Vec is absent and Inter-Vec fails, Intra-Vec provided by the platform is used. If Inter-Vec succeeds, PPOpenCL, as a source-to-source compiler, expresses this by coalescing $f_v$ work-items together (lines 9 – 10) and vectorizing the kernel code in terms of vector types and intrinsic functions for every kernel object with a continuous layout (lines 11 – 15). If needed, some affected kernel objects are padded (with the pads initialized with the existing values of some kernel objects).

**Table 4: Benchmarks (together with their descriptions and input sizes used)**

| Program | Source | Description | Input Size |
|---|---|---|---|
| cutcp | Parboil | Biomolecular Simulation | 96603 atoms |
| lbm | Parboil | Fluid Dynamics | 120x120x150 cells |
| mri-q | Parboil | MRI Reconstruction in non-Cartesian Space | 3x262144 pixels |
| sgemm | Parboil | Matrix Multiply | 1024x992, 1056x992 elements |
| stencil | Parboil | 7-point Stencil | 512x512x64 grids |
| spmv | Parboil | Sparse matrix-vector Multiplication | 146689 elements |
| kmeans | Rodinia | Data Mining | 819200 points x 34 features |
| backprop (bp) | Rodinia | Pattern Recognition | 65537x17 nodes |
| nw | Rodinia | Dynamic Programming | 2048x2048x4 points |
| lavaMD | Rodinia | Molecular Dynamics | 1000 boxes |
| b+tree | Rodinia | B+tree Traversal | 7874 nodes |
| cfd | Rodinia | Computational Fluid Dynamics | 193536 elements |
| gaussian | Rodinia | Gaussian Elimination | 1024x1024 matrix |
| streamcluster (sc) | Rodinia | Dense Linear Algebra | 65536 points |

**Algorithm 4** Holistic Vectorization

1: $tmpFactor['x'] = [SIMDWIDTH]$ // the SIMD width
2: $tmpFactor = ApplyConstraints(tmpFactor, \{'x'\})$
3: $InterFactor = tmpFactor['x']$ // Only $x$ vectorizable
4: $reg_{wi} = EstimateRegUsage(WIICFG.Kernel())$
5: $InterFactor = min(reg_{all}/reg_{wi}, InterFactor)$
6: $ExpFactor = DetectExpVec(WIICFG.Kernel())$
7: $f_v = InterFactor/ExpFactor$
8: **if** $f_v > 1$ **then**
9: $\quad CoarsenThreads(WIICFG.Kernel(), 'x', f_v)$
10: $\quad ChangeNDRange(WIICFG.Host(), 'x', f_v)$
11: $\quad$ **for all** kernel objects $A$ **do**
12: $\quad\quad$ **if** $Continous(A)$ **then**
13: $\quad\quad\quad GenerateSIMD(A, f_v)$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end if**

Figure 9 gives an example illustrating our holistic vectorization, assuming a vector length of four float elements. In the original code given in Figure 9(a), Exp-Vec is present, with two vector lanes already taken. By applying Inter-Vec, PPOpenCL finds that $f_v = 2$. Merging two adjacent work-items gives rise to the optimized code in Figure 9(b).

## 4 EVALUATION

We have implemented PPOpenCL in Clang v3.9.1 and evaluated it on seven platforms listed in Table 3. We have tried all the 11 programs from `Parboil` and all the 21 programs from `Rodinia`. Note that `SPEC ACCEL` [19] consists of 19 programs selected from these two benchmark suites with different inputs. Table 4 lists the 14 programs (with 11 in `ACCEL` [19]) for which PPOpenCL achieves either a positive or negative (i.e., non-1.0x) speedup on at least one platform, together with their input sizes. Note that we will also give the speedups achieved by PPOpenCL not only for these benchmarks but also for the two benchmark suites as a whole.

**Table 3: Seven serial and data-parallel platforms.**

| Platform | | Compute Unit | OpenCL SDK |
|---|---|---|---|
| Name | Processor | | |
| Intel CPUs | Xeon E7- 8830 | 64 cores, 128 threads, 128-bit vector operations | Intel OpenCL SDK |
| AMD CPUs | Opteron | 8 cores, 128-bit vector operations | AMD APP SDK |
| Xeon Phi | Xeon E5-2670 & Xeon Phi | 57 cores, 228 threads, 512-bit vector operations | Intel OpenCL SDK |
| Tilera | TileGX-36 | 36 cores, no vector operations | SNU-SAMSUNG OpenCL [22] |
| SW 26010 | MPE & CPE clusters | 256-bit vector operations | PPOpenCL with Sunway backend |
| NVIDIA GPUs | CPU & Tesla K40c | 15 SMX, 2880 cores | NVIDIA CUDA SDK |
| AMD GPUs | CPU & Radeon HD 7950 | 28 SIMD engines, 1792 stream cores | AMD APP SDK |

For `Parboil`, a program has two versions, `OpenCL_base` for CPUs and `OpenCL_nvidia` for GPUs. For `Rodinia`, a program has only one version. For each baseline version on a platform (`OpenCL_nvidia` for GPUs and `OpenCL_base` for others), PPOpenCL

will produce an optimized version. Both versions will be compiled by the platform-specific SDK. The speedup achieved by PPOpenCL is measured as the execution time of the baseline over the optimized version.

## 4.1 Overall Performance Improvements

Figure 10 shows the speedups achieved by PPOpenCL for the programs given in Table 4. For a striped bar, its corresponding platform's SDK either failed to compile the baseline program or the compiled baseline crashed due to a runtime error. We have not seen a case where this compile-time/runtime error happened to an optimized version but not to its baseline.

Our evaluation shows that PPOpenCL can deliver portable performance. For the 14 programs in Table 4, their speedups are fairly impressive across the seven platforms: 1.22x for Intel CPUs, 2.22x for AMD CPUs, 1.53x for Xeon Phi, 1.50x for Tilera, 2.18x for SW26010, 1.23x for NVIDIA GPUs, and 1.30x for AMD GPUs, with a geomean of 1.55x. PPOpenCL achieves the lowest speedup for NVIDIA GPUs as most of these benchmarks are originally written for NVIDIA GPUs. Note that speedup fluctuations are also an indication for the poor performance portability of the baseline programs across these platforms. Also, `spmv` (the `Opencl_nvidia` version) and `stencil` (the `Opencl_base` version) suffer from some performance losses on GPUs and Xeon Phi, respectively. A more sophisticated cost model may be needed to overcome this limitation in future work.

For the other 5 programs from `Parboil` and the 13 programs from `Rodinia`, which are not listed in Table 4, PPOpenCL has managed to optimize only two from `Rodinia`, `nn` and `lud`, but without any performance impact at all. With all these programs also counted (but without the ones whose baselines cannot be executed correctly, as discussed above), the speedups achieved by PPOpenCL are still impressive: 1.12x for Intel CPUs, 1.49x for AMD CPUs, 1.26x for Xeon Phi, 1.25x for Tilera, 1.52x for SW26010, 1.11x for NVIDIA GPUs, and 1.16x for AMD GPUs.

## 4.2 Individual Optimizations

Figure 11 compares the percentage contributions made by PPOpenCL's three optimizations to the performance improvement of a program on each platform. These results are obtained by gradually introducing the three optimizations, thread reorganization, data layout optimization, and holistic vectorization, in that order (applied).

Holistic vectorization is not profitable on Intel CPUs, Xeon Phi and Tilera. For the former two, Intel SDK applies inter-work-item vectorization by default. So PPOpenCL's vectorization is turned off (Section 3.3.3). For Tilera, vector operations are not supported. Unlike holistic vectorization, thread reorganization and data layout optimization are both significant on all platforms and more application-specific.

Let us analyze two benchmarks in Figure 12:

(1) `kmeans`. Thread reorganization is applied to all platforms, but without any noticeable performance gains.
Data layout optimization is applied to a buffer, named `d_features`, which has a coalesced layout in the baseline program. PPOpenCL leaves this unchanged for GPUs, thus resulting in no gains on NVIDIA and AMD GPUs. For the remaining five platforms, PPOpenCL selects a continuous layout for `d_features`, resulting in varying gains. However, the absolute performance gains for Intel CPUs and Xeon Phi are smaller, as the reuse distances shortened on both are not as impressive. For example, the reuse distance on AMD CPUs (Tilera) is 51200 (204800) for the coalesced layout but drops to 0 for the continuous layout. For both Intel CPUs and Xeon Phi, however, the same reduction is achieved but only from 33 to 15.
As for holistic vectorization, we explained earlier this is not enabled for Intel CPUs, Xeon Phi and Tilera. For SW26010, it was not applied. For AMD CPUs, the baseline could not be vectorized by AMD SDK. PPOpenCL's vectorization is thus profitable. For NVIDIA and AMD GPUs, the performance gains are mainly attributed to the 64-bit and 128-bit vector loads/stores enabled.

(2) `sgemm`. Thread reorganization is now profitable for all the platforms, as some redundant loads for `d_a` and `d_b` have been removed. The absolute gains are less impressive on NVIDIA and AMD GPUs due to the smaller coarsening factors used ($f_c = 2$ on GPUs vs. $f_c = 16$ for the others). Data layout optimization is applied to `d_a` and `d_b`, with a coalesced layout initially in each case. Unlike `d_features` above, this optimization, which is similarly applied to both, is less profitable in percentage terms. Finally, holistic vectorization achieves performance gains on AMD CPUs and SW26010 (as the baseline was not successfully vectorized) and on NVIDIA and AMD GPUs (due to the 64- and 128-bit vector loads/stores enabled).

*4.2.1 Thread Reorganization.* We analyze this using `oclQuasirandomGenerator`, abbreviated here to `ocl`, from [29]. Unlike those from Table 4, `ocl` allows us to demonstrate how PPOpenCL selects different optimization strategies for NVIDIA and AMD CPUs.

This program has a work-group size of $[320, 3, 1]$. Coarsening its $x$ dimension allows some redundant load instructions to be removed. On the other hand, coarsening its $y$ dimension allows some redundant branch instructions to be removed. For each of the seven platforms considered, Figure 13 depicts the coarsening factor (and its corresponding coarsening dimension) selected by PPOpenCL, together with the performance speedup variations with these coarsening factors. For all the seven platforms, PPOpenCL has succeeded in picking the best coarsening factor except for Tilera.

For Intel CPUs and Xeon Phi, coarsening the $x$ dimension potentially degrades performance, as it makes it harder to vectorize the work-items along $x$. For AMD CPUs and SW26010, the performance increases as the coarsening factor along $x$ increases, due to eliminated load instructions. Tilera is expected to follow the same trend except for 4 ($x$). For NVIDIA GPUs, coarsening the $y$ dimension is better, since this eliminates some branch instructions and thus reduces warp divergence. For AMD GPUs, coarsening the $x$ dimension is better, as eliminating some redundant loads is more performance-critical. However, any coarsening factor larger than 2 along the $x$ dimension degrades performance as the resulting work-group size will no longer be a multiple of 32.

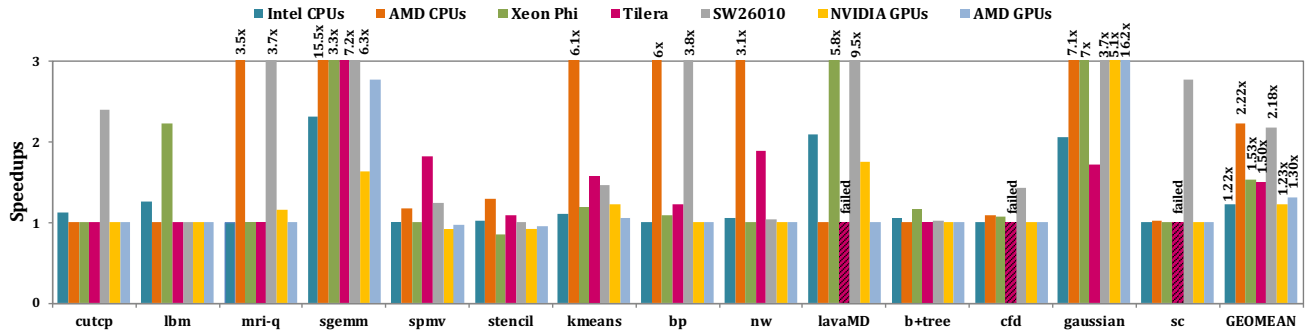



**Figure 10: Performance speedups on seven platforms.**

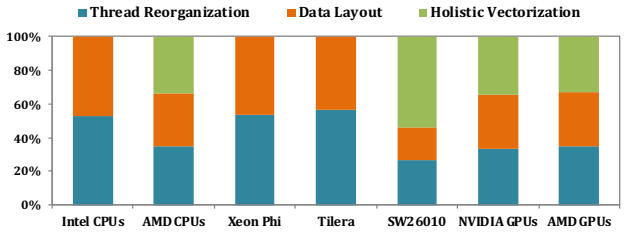



**Figure 11: Percentage contributions of three optimizations.**

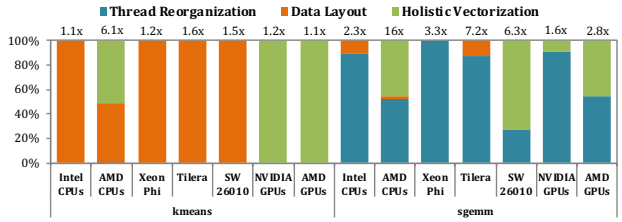



**Figure 12: Percentage contributions of three optimizations on two benchmarks, `kmeans` and `sgemm`. For convenience, their speedups from Figure 10 are duplicated.**

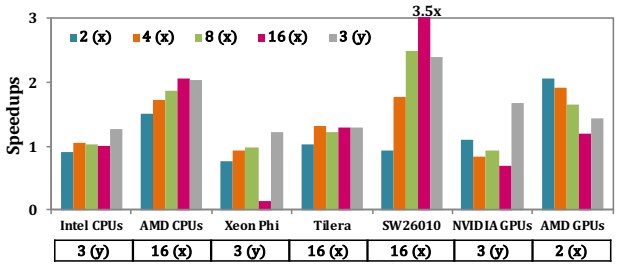



**Figure 13: Speedups with the coarsening factors along the $x$ or $y$ dimensions of `ocl` (selected by PPOpenCL).**

*4.2.2 Data Layout Optimization.* We examine this optimization using `lbm` from `Parboil`. Each work-item operates on a set of 20 data elements associated with a cell in a 3-D mesh. In a continuous layout (adopted by the `opencl_base` version), the 20 data elements for a cell are stored together. In a coalesced layout (adopted by the `opencl_nvidia` version), the data elements with the same attribute from all the cells are stored together.

**Table 5: Cache Behavior of `lbm` (by OProfile/Perf/Vtune).**

| Platform | Continuous Layout | Coalesced Layout |
|---|---|---|
| Intel CPUs (L1_DCACHE_LOAD_MISS) | 8.02E+10 | **5.47E+10** |
| Xeon Phi (L1 miss rate) | 0.335 | **0.151** |
| AMD CPUs ((DATA_CACHE_MISSES) | **3.63E+11** | 1.81E+12 |
| Tilera (READ_MISS) | **6.58E+12** | 6.74E+12 |

For `opencl_base`, PPOpenCL retains its continuous layout for AMD CPUs and Tilera but switches to a coalesced layout for Intel CPUs and Xeon Phi (based on reuse distance analysis), achieving 1.25x on Intel CPUs and 2.21x on Xeon Phi (Figure 10). As shown in Table 5, this optimization has succeeded in reducing the cache miss rate on each platform.

*4.2.3 Holistic vectorization.* We study this optimization using `mri-q` from `Parboil` for AMD CPUs, Xeon Phi and NVIDIA GPUs. As the baseline program is not vectorized, PPOpenCL has vectorized its work-items by a factor of 4 for AMD CPUs and NVIDIA GPUs. As for Xeon Phi, this vectorization is not performed, as it is done by Intel SDK (Section 3.3.3). We show that performing this optimization by PPOpenCL can be counter-productive.

Figure 14 relates the speedups achieved with the number of vector arithmetic instructions (varith_ins) and the number of scalar and vector loads (load_ins), measured by using *nvprof* for NVIDIA GPUs, *OProfile* for AMD CPUs and *VTune* for Xeon Phi (normalized to the baseline).

For NVIDIA GPUs, arithmetic computations cannot be vectorized but memory accesses can. With `load_ins` reduced to a quarter of that in the baseline, a speedup of 1.15x is obtained. For AMD

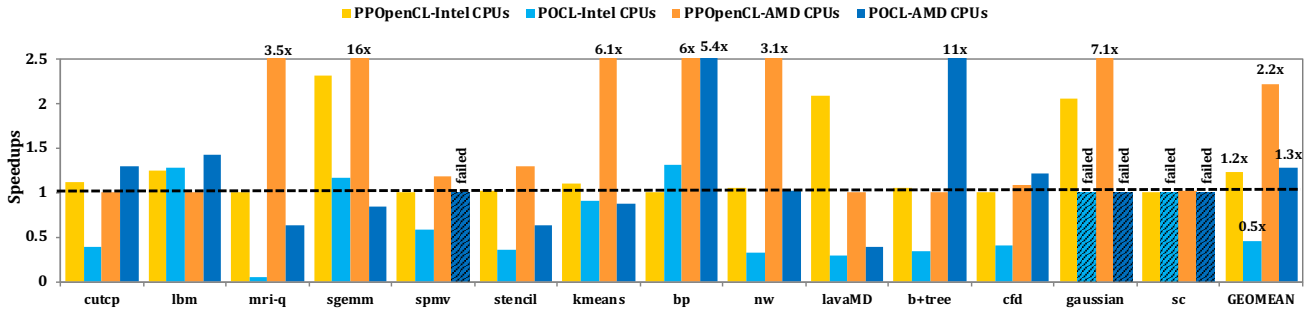



**Figure 15: Comparing PPOpenCL and POCL.**

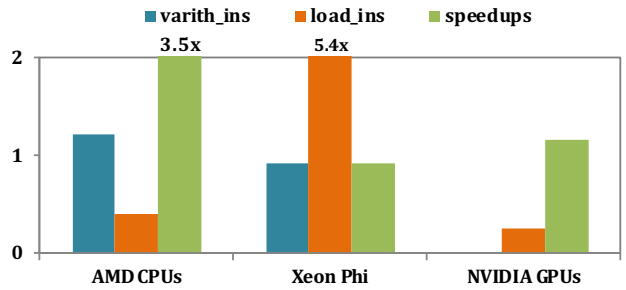



**Figure 14: Effects of vectorization for `mri-q`.**

CPUs supporting vector operations, `varith_ins` increases but `load_ins` decreases, a speedup of 3.15x is observed. For Xeon Phi, for which Intel SDK always applies inter-work-item vectorization to the baseline, vectorizing it earlier by PPOpenCL affects `varith_ins` slightly, but increases `load_ins` substantially due to register spills, resulting a performance loss of about 10%. Hence, PPOpenCL is designed not to perform holistic vectorization for Intel CPUs and Xeon Phi (Section 3.3.3).

## 4.3 PPOpenCL vs. POCL

We have compared PPOpenCL with POCL [16] on Intel and AMD CPUs, since these are the stable platforms supported by POCL currently. Figure 15 compares both using the benchmarks in Table 4. PPOpenCL outperforms POCL with a geomean of 2.4x (1.7x) on Intel (AMD) CPUs. Under POCL, `gaussian` and `sc` had seg-faults (marked by striped bars).

For Intel CPUs, POCL yields notable performance slowdowns in most programs, with speedups only for `lbm`, `sgemm` and `bp`, outperforming Intel SDK, on average. POCL executes the work-items in a work-group in scalar mode, while Intel SDK vectorizes them (Figure 5(c)). For AMD CPUs, POCL adopts a similar optimization strategy as AMD SDK, resulting in more speedups outperforming AMD SDK, on average (due to mainly the performance gain for `b+tree` by POCL).

Unlike POCL, which focuses on optimizing kernel codes only, PPOpenCL is a source-to-source compiler by taking advantage of a platform-specific OpenCL implementation to avoid performance drops. PPOpenCL outperforms POCL in 12 (10) programs on Intel CPUs (AMD CPUs), since it optimizes both host and kernel thread codes simultaneously. In the case of `b+tree`, POCL significantly outperforms PPOpenCL on AMD CPUs. By applying aggressive loop unrolling, POCL has successfully vectorized the operations on complex data structures involving branch instructions across 256 adjacent work-items. However, PPOpenCL failed in its vectorization attempt due to the lack of a powerful dependence analysis for complex data structures. Similarly, AMD SDK (as in the case of Intel SDK) did not succeed here, either.

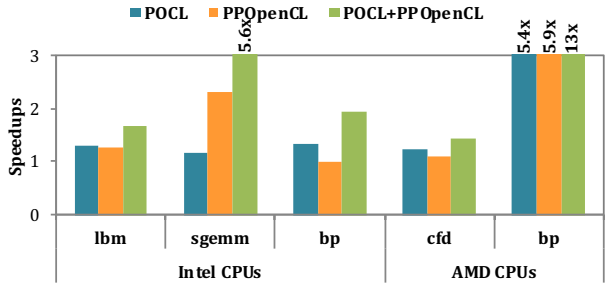



**Figure 16: Combining PPOpenCL and POCL.**

We can obtain the best of the two worlds by combining PPOpenCL and POCL. For the benchmarks at which both achieve positive speedups (Figure 16), PPOpenCL + POCL is superior over either alone, as PPOpenCL applies kernel optimizations that require the host code information and POCL applies kernel optimizations during code generation.

## 4.4 PPOpenCL vs. OpenACC

OpenACC [27, 30, 31] is another cross-platform programming framework, which is less performance-competitive than OpenCL but more performance portable. We make this comparison using `reduction`, with its OpenACC version from `NAS_SHOC_OpenACC` [14] and its OpenCL version from `SHOC` [6], on Intel CPUs, AMD CPUs and NVIDIA GPUs. The PGI [33] tool chains for OpenACC are used on these platforms (with OpenACC as the baseline).

Figure 17 gives our results. PPOpenCL outperforms OpenACC by 1.02x on Intel CPUs, 1.12x on AMD CPUs, and 1.19x on NVIDIA GPUs. There are significant performance differences on AMD CPUs.

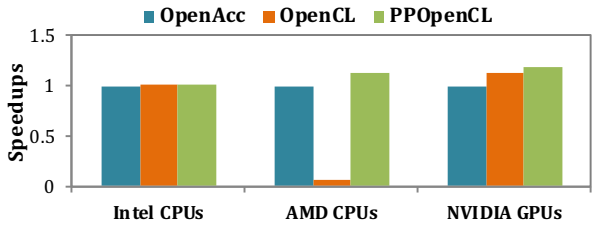



**Figure 17: Comparing PPOpenCL and OpenACC.**

OpenCL is significantly slower on AMD CPUs than on the other two platforms. The OpenCL version utilizes a coalesced layout and launches a large number of work-items, resulting in poor cache utilization and excessive runtime overhead on AMD CPUs. By applying data layout optimization and holistic vectorization, PPOpenCL achieves a speedup of 16.5x over OpenCL.

## 5 RELATED WORK

*Optimization for Multiple Accelerators.* There are many efforts on optimizing OpenCL/CUDA programs on GPUs [3–5, 12, 15, 35, 36, 48, 51], CPUs [17, 20], and FPGAs [9, 18, 50, 59], which are mostly restricted to one accelerator. The three optimizations supported by PPOpenCL are also considered elsewhere [24, 25, 38], except that PPOpenCL provides a generic compiler solution to facilitate host-kernel fused optimizations. Kernel fusion [10, 49], which is the closest to our work, fuses kernels to improve performance rather than its portability. Others [23, 53] optimize OpenCL/CUDA programs for multiple NVIDIA and AMD GPUs. In contrast, PPOpenCL optimizes both host and kernel thread codes simultaneously to improve performance portability by modeling explicitly the platform execution order across the work-items.

*Extended Control/Data Flows.* There are some efforts on constructing inter-thread control/data flows [1, 42, 55] to facilitate parallelization, communication optimization, or task scheduling. They focus on inter-thread control and data dependencies. In contrast, PPOpenCL focuses on incorporating platform-specific execution orders into WII-CFG.

*Performance Portability for Heterogeneous Architectures.* Some efforts exist for improving performance portability for specific applications [37, 45] or designing programming languages to support different optimizations on different platforms [2, 44]. They focus on architectural differences while PPOpenCL stresses on modeling platform-specific execution orders for the work-items in a work-group.

POCL [16] is a performance-portable OpenCL implementation. Its core is a kernel compiler that can exploit the data parallelism in OpenCL programs on multiple platforms with different parallel execution models. HPVM [43] is a program representation designed to enable cross-platform performance portability for parallel hardware, by virtualizing the parallel execution behavior and the parallel hardware ISAs. It treats work-items as nodes in a data flow graph and handles the work-items from each kernel uniformly for optimization purposes. Like POCL and HPVM, PPOpenCL considers the multiple work-items in a work-group when applying optimizations to improve performance portability. Unlike POCL and HPVM, however, PPOpenCL can expose more optimization opportunities by performing its optimizations in both the host and kernel thread codes simultaneously.

OpenACC [27, 30, 31] is another cross-platform heterogeneous programming model, which is less performance-competitive but more performance portable than OpenCL [40]. An OpenACC program is written as a serial C/Fortran program annotated with pragmas, with no separate host and kernel programs as in an OpenCL/CUDA program. Such a unified programming model and PPOpenCL will benefit from each other in several ways. First, a unified programming model will simplify our analysis in identifying aliased host and kernel objects, since the data are unified. Second, a unified programming model provides more opportunities for applying a more precise alias analysis to the unified code, before the host and code codes are separated. Finally, our approach can apply to a unified programming model, by treating each parallel-loop as a kernel. We can continue to use WII functions to specify the execution orders of iterations in parallel-loops, thread re-organization and holistic vectorization to optimize parallel-loops, and data layout optimizations to improve locality across the boundary of pragmas.

## 6 CONCLUSION

In this paper, we have introduced a source-to-source OpenCL compiler, PPOpenCL, to improve cross-platform performance portability for OpenCL programs, by fusing the host and kernel thread codes of an OpenCL program together. We have implemented PPOpenCL in Clang and conducted a fairly extensive evaluation in terms of a set of commonly used OpenCL benchmarks on seven representative platforms. Our experimental results demonstrate that PPOpenCL improves the state of the art by delivering better portable performance.

## ACKNOWLEDGMENTS

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National Key Research and Development Program of China (2017YFB0202002), the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485, 61872043), and an Australian Research Council (ARC) Grant DP180104069.

## REFERENCES

[1] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. 2014. Compiler Techniques for Massively Scalable Implicit Task Parallelism. In *Proceedings of the 26th International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE, New Orleans, LA, USA, 299–310.

[2] Li-Wen Chang, Izzat El Hajj, Christopher Rodrigues, Juan Gómez-Luna, and Wen-mei Hwu. 2016. Efficient Kernel Synthesis for Performance Portable Programming. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*. IEEE, Taipei, Taiwan, 12:1–12:13.

[3] Huimin Cui, Lei Wang, Jingling Xue, Yang Yang, and Xiaobing Feng. 2011. Automatic Library Generation for BLAS3 on GPUs. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 255–265.

[4] Huimin Cui, Jingling Xue, Lei Wang, Yang Yang, Xiaobing Feng, and Dongrui Fan. 2012. Extendable pattern-oriented optimization directives. *ACM Transactions on Architecture and Code Optimization* 9, 3 (2012), 14.

[5] Huimin Cui, Qing Yi, Jingling Xue, and Xiaobing Feng. 2013. Layout-Oblivious Compiler Optimization for Matrix Computations. *Acm Transactions on Architecture and Code Optimization* 9, 4 (2013), 1–20.

[6] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*. ACM, Pittsburgh, Pennsylvania, USA, 63–74.

[7] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, San Diego, California, USA, 245–257.

[8] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Computing*. 38, 8 (Aug. 2012), 391–407.

[9] Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, and Jim Wu. 2016. Optimizing OpenCL Applications on Xilinx FPGA. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL '16)*. ACM, Vienna, Austria, 5:1–5:2.

[10] Jiri Filipovic, Matus Madzin, Jan Fousek, and Ludek Matyska. 2015. Optimizing CUDA Code By Kernel Fusion:Application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[11] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, et al. 2016. The Sunway TaihuLight supercomputer: system and applications. *SCIENCE CHINA Information Sciences* 59, 7 (2016), 072001:1–072001:16.

[12] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David Kaeli. 2017. TwinKernels: An Execution Model to Improve GPU Hardware Scheduling at Compile Time. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE, Austin, USA, 39–49.

[13] Khronos Group. 2018. OpenCL Overview. https://www.khronos.org/opencl/

[14] OpenACC User Group. 2017. NAS SHOC OpenACC 2.5. https://github.com/OpenACCUserGroup/openacc-users-group/tree/master/Contributed_Sample_Codes/NAS_SHOC_OpenACC_2.5

[15] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan. 2015. Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*. ACM, 143–153.

[16] Pekka Jääskeläinen, Carlos Sánchez Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. Pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*. 43, 5 (Oct. 2015), 752–785.

[17] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-dimensional, Winograd-based Convolution for Manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vienna, Austria, 109–123.

[18] Zheming Jin and Hal Finkel. 2018. Performance-oriented Optimizations for OpenCL Streaming Kernels on the FPGA. In *Proceedings of the International Workshop on OpenCL (IWOCL '18)*. ACM, Oxford, United Kingdom, 1:1–1:8.

[19] Guido Juckeland, William C. Brantley, Sunita Chandrasekaran, et al. 2014. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *Proceedings of 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'14)*. Springer, New Orleans, LA, USA, 46–67.

[20] Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta, and Wen-Mei Hwu. 2015. Locality-centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE/ACM, San Francisco, California, 257–268.

[21] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156.

[22] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, et al. 2010. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *Proceedings of the 19th ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. IEEE, Vienna, Austria, 193–204.

[23] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic Optimization of Thread-coarsening for Graphics Processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, Edmonton, AB, Canada, 455–466.

[24] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, Denver, Colorado, USA, Article 11, 11 pages.

[25] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2016. Automatic data layout generation and kernel mapping for CPU+GPU architectures. In *Proceedings of the 21st International Conference on Compiler Construction (CC '16)*. ACM, Barcelona, Spain, 240–250.

[26] Kathryn S Mckinley, Steve Carr, and Chauwen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems* 18, 4 (1996), 424–453.

[27] Douglas Miles, David Norton, and Michael Wolfe. 2014. Performance Portability and OpenACC. In *Proceedings of Cray Users Group Meeting (CUG '14)*. Lugano, Switzerland, 1–8.

[28] NVIDIA. 2018. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[29] NVIDIA. 2018. NVIDIA OpenCL SDK Code Samples. https://developer.nvidia.com/opencl

[30] NVIDIA. 2018. Performance Portability from GPUs to CPUs with OpenACC. https://devblogs.nvidia.com/performance-portability-gpus-cpus-openacc/

[31] OpenACC. 2018. OpenACC Specification. https://www.openacc.org/specification

[32] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. 2013. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing*. 73, 11 (Nov. 2013), 1439–1450.

[33] PGI. 2018. PGI Accelerator Compilers with OpenACC Directives. https://www.pgroup.com/resources/accel.htm

[34] James Price and Simon McIntosh-Smith. 2017. Analyzing and Improving Performance Portability of OpenCL Applications via Auto-tuning. In *Proceedings of the 5th International Workshop on OpenCL (IWOCL '2017)*. ACM, Toronto, Canada, Article 14, 4 pages.

[35] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, Haifa, Israel, 99–111.

[36] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vienna, Austria, 168–182.

[37] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance Portable GPU Code Generation for Matrix Multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU '16)*. ACM, Barcelona, Spain, 22–31.

[38] Ingo Wald Roland Leiba, Sebastian Hack. 2012. Extending a C-like Language for Portable SIMD Programming. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New Orleans, Louisiana, USA, 65–74.

[39] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Tibor Grasser, and Ansgar Jüngel. 2014. Performance Portability Study of Linear Algebra Kernels in OpenCL. In *Proceedings of the International Workshop on OpenCL (IWOCL '14)*. ACM, Bristol, UK, Article 8, 11 pages.

[40] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S. Vetter. 2014. Evaluating Performance Portability of OpenACC. In *Proceedings of 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC '14)*. Springer, Hillsboro, OR, USA, 51–66.

[41] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. 2013. Automatic OpenCL work-group size selection for multicore CPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE, Edinburgh, UK, 387–397.

[42] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Austin, TX, USA, 81:1–81:12.

[43] Prakalp Srivastava, Maria Kotsifakou, and Vikram S. Adve. 2016. HPVM: A Portable Virtual Instruction Set for Heterogeneous Parallel Systems. *CoRR* abs/1611.00860 (2016). arXiv:1611.00860

[44] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, Vancouver, BC, Canada, 205–217.

[45] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16)*. ACM, Pittsburgh, Pennsylvania, USA, 15:1–15:10.

[46] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and W.W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. University of Illinois at Urbana-Champaign.

[47] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 265–266.

[48] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017)*. ACM, Barcelona, Spain, 11–20.

[49] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE, New Orleans, LA, USA, 191–202.

[50] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. 2017. Energy Efficient Scientific Computing on FPGAs Using OpenCL. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, Monterey, California, USA, 247–256.

[51] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An Open-source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, Barcelona, Spain, 105–116.

[52] Jingling Xue and Jens Knoop. 2006. A Fresh Look at PRE as a Maximum Flow Problem. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*. 139–154.

[53] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. 2012. A Unified Optimizing Compiler Framework for Different GPGPU Architectures. *ACM Transactions on Architecture and Code Optimization*. 9, 2, Article 9 (June 2012), 33 pages.

[54] Yao Zhang, Mark Sinclair II, and Andrew A. Chien. 2013. Improving Performance Portability in OpenCL Programs. In *Proceedings of the 28th International Supercomputing Conference (ISC '13)*. Springer, Leipzig, Germany, 136–150.

[55] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. 2010. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT '10)*. ACM, Vienna, Austria, 169–180.

[56] Hao Zhou and Jingling Xue. 2016. A Compiler Approach for Exploiting Partial SIMD Parallelism. *ACM Trans. Archit. Code Optim.* 13, 1, Article 11 (March 2016), 26 pages. https://doi.org/10.1145/2886101

[57] Hao Zhou and Jingling Xue. 2016. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/2854038.2854054

[58] Hans Zima and Barbara Chapman. 1991. *Supercompilers for Parallel and Vector Computers*. ACM, New York, NY, USA.

[59] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE, Salt Lake City, UT, USA, 409–420.