2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Performance Traps in OpenCL for CPUs

Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu
Parallel and Distributed Systems Group
Delft University of Technology
Delft, The Netherlands
Email: {j.shen, j.fang, h.j.sips, a.l.varbanescu}@tudelft.nl

Abstract*—With its design concept of cross-platform portability, OpenCL can be used not only on GPUs (for which it is quite popular), but also on CPUs. Whether porting GPU programs to CPUs, or simply writing new code for CPUs, using OpenCL brings up the performance issue, usually raised in one of two forms: “OpenCL is not performance portable!” or “Why using OpenCL for CPUs after all?!”. We argue that both issues can be addressed by a thorough study of the factors that impact the performance of OpenCL on CPUs. This analysis is the focus of this paper. Specifically, starting from the two main architectural mismatches between many-core CPUs and the OpenCL platform—parallelism granularity and the memory model—we identify eight such performance “traps” that lead to performance degradation in OpenCL for CPUs. Using multiple code examples, from both synthetic and real-life benchmarks, we quantify the impact of these traps, showing how avoiding them can give up to 10 times better performance. Furthermore, we point out that the solutions we provide for avoiding these traps are simple and generic code transformations, which can be easily adopted by either programmers or automated tools. Therefore, we conclude that a certain degree of OpenCL inter-platform performance portability, while indeed not a given, can be achieved by simple and generic code transformations.

***Index Terms*—OpenCL, Many-core CPUs, Performance portability**

I. Introduction

As multi- and many-core processors grew in both complexity and diversity, the community of hardware and software vendors have proposed an open standard programming model for parallel computing, called OpenCL (Open Computing Language) [1]. The model is designed as a virtual computing platform across CPUs, GPUs, and other processors alike. Users design and develop applications for this virtual platform, and the parallelized applications can be run on all processors[1] that support it. In OpenCL, programmers are able to manage OpenCL initialization, data transfers and kernel execution by a set of language extensions and runtime APIs.

Originating from the GPGPU world, OpenCL has a lot of similarities with CUDA (Compute Unified Device Architecture) [2], and therefore it keeps gaining popularity for GPU programming since 2009, when the first standard was published [3]. However, CPUs remain of interest: all major CPU vendors (e.g., Intel, AMD, ARM) have already released their own OpenCL SDKs, and they are continuously improving these implementations, aiming to achieve better performance on their CPUs. Furthermore, recent comparative studies of programming models [4], [5] have shown that the performance OpenCL applications achieve on many-core CPUs can be comparable to that obtained using more traditional models, such as OpenMP.

As OpenCL has been widely used on GPUs, there is already a large collection of OpenCL GPU codes available [6]–[8]. We argue that these codes are valuable resources *also* for many-core CPUs given their functional correctness and large scale parallelism (both enabled by the OpenCL model). They are performing, in most cases, better than their sequential counterparts. In other words, these codes provide parallel CPU implementations to be tested and/or tuned, without the additional effort of developing OpenMP [9], Pthreads [10], or TBB/ArBB [11], [12] versions. While we do not claim that the OpenCL implementations are as good as these specifically designed parallel codes, we argue that, for a given application, its *available* OpenCL code can show one of two things: (1) surprisingly good performance, when the parallelization strategy and systematic, generic optimizations fit the application, or (2) a clear indication that the large-scale fine-grained parallelization does not match the given application. Either way, programmers eventually gain a better understanding of the application behavior for the small price of applying a few systematic code transformation techniques and straightforward performance measurement/benchmarking of the OpenCL code. Furthermore, having both the GPU and CPU codes available for the same application allows more flexibility in the execution scenarios, especially for cases when multiple applications compete for the resources of the same machine [13], [14].

Starting from the hypothesis that (available) OpenCL code can and should run well on CPUs, this paper focuses on answering one essential question: what are the factors that programmers need to pay attention to when writing OpenCL code for CPUs? We believe that the answer to this question will actually provide the OpenCL community with solutions to improve OpenCL for CPUs, be it at the model, the compiler, or the coding style level, ultimately delivering an acceptable degree of inter-platform performance portability.

To answer our main research question, we start with an analysis of the “visible” mismatches between the OpenCL platform and the CPUs (Section II), which gives us clues to the performance hit causes. Using a set of GPU-style benchmarks (some ported, some specifically written), we expose several performance problems, called “traps” (because they can be easily overlooked while porting applications). By applying successive, systematic code transformations and recording their impact on performance, we are able to both *quantify* the importance of these traps and *show generic solutions* to avoid them (Section III). We have deliberately used already available transformation techniques: by keeping things simple

[1]Currently, Apple, Intel, AMD, NVIDIA, IBM, and ARM have processors/devices that support OpenCL.

and generic, we believe both programmers (for now) and automated tools (for the near future) can implement the same transformations with little programming effort. Based on these observations, we provide a short list of empirical guidelines (Section IV) for OpenCL programmers to take into account when porting OpenCL code for CPUs.

II. Looking for Traps

According to its specification [15], OpenCL shares the core parallelism approach with CUDA, which is designed for NVIDIA GPUs. Consequently, there are visible mismatches between the OpenCL platform and the CPUs, with significant effects on performance. In this section, we list these mismatches, analyze their consequences, and show how we use them to identify performance glitches in OpenCL on CPUs.

A. Mismatches between the OpenCL platform and the CPUs

a) Parallelism granularity: The OpenCL platform consists of a host and one or more OpenCL compute devices (see Fig. 1). A compute device is divided into one or more compute units (CUs); CUs are further divided into one or more processing elements (PEs). PEs perform the computations (compute kernels). An instance of a compute kernel is called a `work-item`, and multiple work-items can be grouped into `work-groups`.

OpenCL PEs are “slim” cores. Each PE typically takes charge of one work-item and completes a simple task. GPU hardware cores are small and simple, matching the OpenCL platform well. CPU cores are “fat” cores, designed to perform complicated tasks with the support of dedicated hardware, a mismatch for the OpenCL slim PEs.

The assumption of OpenCL is that there are a lot of PEs in each CU, so that massive parallelism can be achieved. As GPUs have hundreds of cores, the OpenCL platform can be directly mapped to a GPU platform. The number of CPU cores is a lot smaller, and mapping the fine-grained OpenCL PEs onto CPUs is not as natural.

In addition, CPUs have SIMD (Single Instruction Multiple Data) units. With the support of SIMD instruction sets and vector registers, data elements can be packed into vector data to run simultaneously on the SIMD hardware. OpenCL does support vector data, but the OpenCL platform does not have a corresponding SIMD unit. Instead, the OpenCL CPU programmers and/or compilers (different for each provider) need to do the packing of data/work-items into SIMD-ready data and instructions. This code vectorization is of less importance for (NVIDIA) GPUs, as GPUs already work in SIMD/SIMT mode.

b) Memory model: On GPUs, the host and the devices are physically independent of each other, having separate memory spaces. In order to complete a parallel task, some forms of explicit data transfers between the host and the device are required. When mapping OpenCL on CPUs, the host and the device are the same CPU, sharing the same memory space. Then, the explicit host to device (H2D) and device to host (D2H) data transfers can become unnecessary.

In terms of memory access patterns, the OpenCL platform imposes no restrictions, so it is up to the device drivers and compilers to make their choices. In this context, the typical memory coalescing for GPUs is orthogonal with the cache-friendly accesses in CPUs. Therefore, OpenCL GPU codes that explicitly use memory coalescing will suffer performance hits on CPUs.

OpenCL local memory is a memory region designed to be shared by all work-items of a work group. This memory region is implemented as an on-chip memory on GPUs, being much faster than the off-chip global memory. Therefore, GPUs take advantage of local memory to improve performance. CPUs do not have a special physical memory designed as local memory, which means that all memory objects in local memory are mapped into sections of global memory.

B. Consequences

The parallelism granularity mismatch leads to OpenCL performance degradation on CPUs because of two main reasons. First, applying a fine-grained parallelization restricts CPU cache utilization, as one work-item, running on one hardware core/thread, only processes a single point or a small proportion of the dataset. Second, executing scalar OpenCL code cannot make use of the SIMD units, thus wasting CPU computing potential. Additionally, the unvectorized code can also cause lower CPU cache utilization and affect performance.

The memory model mismatch also has a negative effect on performance. First, if programmers still use the default data transfer method as they would use on GPUs, or if an OpenCL program is ported directly from CUDA, the performance will be affected by the unnecessary data transfers between the host and the device. Second, the impact of the GPU-like coalesced memory accesses is significant for problems with a 2-dimensional index space because each work-item accesses a column of data elements, which are not adjacent, and data locality is affected. Third, the successful use of local memory on GPUs can mislead programmers into applying the same technique on CPUs. As on CPUs local memory is, in fact, global memory, its use may not provide further performance gain, but rather has a negative impact on performance by adding extra overheads.

C. Finding and quantifying traps

We use the mismatch analysis in Section II-A as a starting point in our search. By doing so, we insure that, even if we don’t find *all* the performance traps, we do identify the ones with very important impact.

After identifying the performance traps, we need to further prove that they indeed impact performance. To do so, we first put together a set of applications from synthetic benchmarks and the Rodinia benchmark suite [8]. The synthetic benchmarks are for GPUs, so the potential problems do manifest. The Rodinia benchmark suite has applications written in CUDA, OpenMP, and OpenCL. As the OpenCL versions are translated from CUDA, they are prone to expose the performance traps we are searching for.

Second, we apply code transformations to these applications and quantify how much influence they have. These transformations are not done randomly. Instead, we check whether the CPU-OpenCL platform mismatches have affected the code, and when needed, we apply the CPU-friendly transformations aiming for performance improvement.

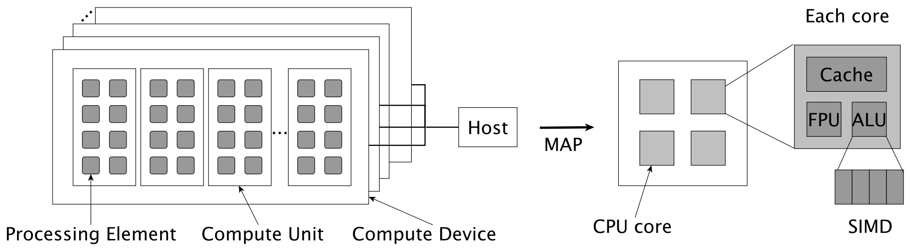


Fig. 1. The parallelism granularity mismatch between the OpenCL platform and the CPUs.

Finally, as the Rodinia benchmark suite also includes OpenMP implementations targeting the CPU platforms, we use the OpenMP experimental results as representative of the performance that ordinary CPU programmers can achieve using a coarse-grained parallelism model (i.e., OpenMP). We also use the performance achieved by OpenMP as an indicator on whether the OpenCL code can be further improved to achieve better performance[2].

III. Performance Traps

In this section, we present a detailed investigation of the performance traps and quantify their impacts.

A. Experimental set-up

We use Intel Xeon E5620 2.40GHz hyper-threaded [16] Dual quad-core (8 cores, 16 hardware threads) as our main hardware platform. Each core has 32KB L1 Data cache and 256 KB L2 cache shared between two hardware threads of the core. The 12MB L3 cache is shared by all the cores. The OpenCL kernels are compiled with two OpenCL compilers: (1) Intel's OpenCL (OCL) SDK 1.5 and (2) AMD Accelerated Parallel Processing (APP) SDK 2.6. The host side environment is GCC version 4.4.6. We further validate our findings on the Intel Xeon X5650 Dual six-core and the AMD Magnycours.

B. H2D and D2H

H2D (host to device) and D2H (device to host) are required memory transfers for OpenCL. On CPUs, real copying is typically not necessary (see Fig. 2), but it can be "forgotten" when porting GPU code.

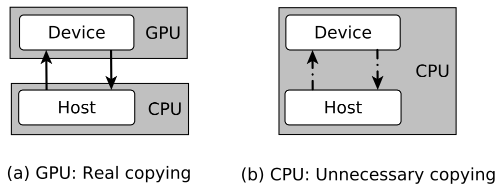


Fig. 2. H2D and D2H data transfers on GPUs and CPUs. Solid arrows represent real copying. Dotted arrows represent unnecessary copying.

[2]In fact, we choose OpenMP as a performance prediction/estimation of what the application itself can achieve in terms of performance. Even if the OpenMP versions are not fully optimized, if they achieve better performance than OpenCL, OpenCL should be further tuned.

We run the HDCOPY benchmark to see how much performance can be lost in this trap. In HDCOPY, a kernel receives an array from the host, performs an operation on each data element, and transfers the output back to the host. Data transfers are performed in the "normal" way by using `clEnqueueWriteBuffer` for H2D and `clEnqueueReadBuffer` for D2H. Table I shows the execution time of each OpenCL section. We notice that the two data transfer sections take a considerable proportion of the total execution time, especially when the amount of data transferred is large.

TABLE I
Execution time (ms) per section and the fraction (%) of H2D and D2H.

	Intel OCL SDK 1.5				AMD APP SDK 2.6			
Dataset	H2D	Kernel	D2H	%	H2D	Kernel	D2H	%
512×512	49.1	26.4	27.0	74.2%	41.7	39.3	42.6	68.2%
1K×1K	202.3	66.4	121.3	83.0%	123.5	88.6	110.2	72.5%
2K×2K	824.0	200.9	489.1	86.7%	485.0	330.7	474.0	74.4%

We then apply the zero copy technique to avoid the unnecessary copying. This is done by changing the explicit copy functions to `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject`. The `map` and `unmap` functions behave like a token. After `map`, the right to read and write a memory buffer is passed to the host. When the host finishes memory accessing, the `unmap` function makes this memory buffer available for the kernel access again. Because no runtime data transfers are performed after applying the transformation, the H2D and D2H time are largely reduced (see Table II.). The overall speedup is 2.5× - 6×.

TABLE II
Execution time (ms) differences between before and after changing the copy method.

	Intel OCL SDK 1.5			AMD APP SDK 2.6		
Dataset	Total$_{before}$	Total$_{after}$	Saved	Total$_{before}$	Total$_{after}$	Saved
512×512	102.5	40.0	58.7	123.7	49.7	63.5
1K×1K	390.1	75.2	303.7	322.3	99.6	204.4
2K×2K	1514.0	247.5	1281.8	1289.6	285.2	925.0

Total represents the total execution time of H2D, kernel, and D2H sections.
Saved represents the amount of time saved in H2D and D2H sections.

According to the above results, H2D and D2H have a significant impact on CPU performance: If the "normal" data transfers are taken for granted, we waste a lot of time. In contrast, one should always consider using zero copy as summarized in the Listing 1.

Listing 1. A general zero copy method

```
// create a memory buffer
S1: buf=clCreateBuffer(size_t size);
// map the memory buffer to the host for H2D or D2H
S2: host_ptr=clEnqueueMapBuffer(buf, CL_MAP_WRITE/READ);
// use the host_ptr to set input data (H2D) or get output data (D2H)
S3: host_side_code(host_ptr)
// unmap the memory buffer and pass the buffer access to the kernel
S4: clEnqueueUnmapMemObject(buf, host_ptr);
// kernel execution
S5: clEnqueueNDRangeKernel(buf, ...);
// release the memory buffer
S6: clReleaseMemObject(buf);
```

C. *Memory access patterns*

When a kernel processes a 2D dataset element by element, the performance is likely to suffer a lot from using the wrong access pattern: row-major or column-major (see Fig. 3). GPUs require column-major order to enable global memory loads and stores to be coalesced into more effective transactions. CPUs require row-major order to preserve the cache locality within each thread.

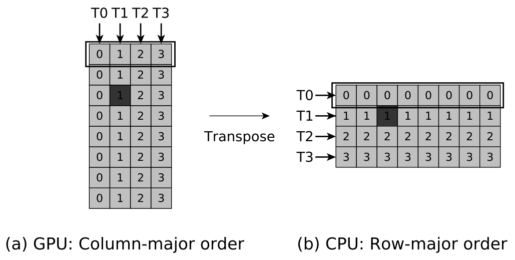


Fig. 3. Memory access patterns on GPUs and CPUs. T0 - T4 represent work-items.

We find this problem in StreamCluster (SC) and K-means (KM) from the Rodinia benchmark suite. These two benchmarks are clustering applications used in data mining. The core in both SC and KM is a N-dimensional Euclidean distance calculation between two points. In the original implementation, the value of each dimension is stored in a column one by one, and therefore each work-item processes one column of data elements. We need to apply a straightforward transposition of the dataset to allow the access pattern to become CPU-friendly.

Fig. 4 shows the impact of this transposition. We see that the row-major version always delivers better performance than the column-major one. The performance gain is around 4$\times$ in KM and 10$\times$ in SC. We further compare the OpenCL implementation and the OpenMP implementation which natively uses the row-major access. With the same row-major order, it is possible for OpenCL to have equal or even better performance than OpenMP.

Overall, row-major and column-major memory access patterns on CPUs are critical to cache locality and in consequence to performance. The impacts of using these two patterns on CPUs and GPUs are orthogonal. When programming directly on CPUs, row-major is the right choice. But porting code from GPUs to CPUs needs examination to determine whether the row-major pattern is used everywhere, and/or it requires transposition.

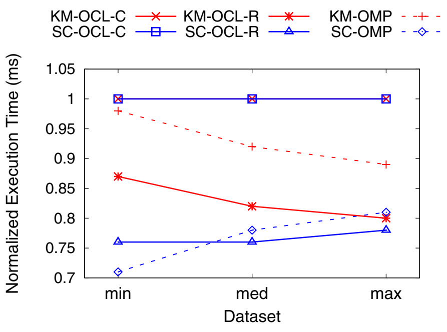


Fig. 4. Normalized execution time of KM and SC. In the OpenCL implementation, C represents the column-major order, R represents the row-major order. In the OpenMP implementation, the row-major order is by default. KM uses datasets of 200K, 482K, and 800K points with 34 dimensions. SC uses datasets of 16K, 32K, and 64K points with 256 dimensions.

D. *Local memory usage*

The use of local memory is a typical optimization on GPUs to enable faster data access and data sharing within a work-group. These benefits are also of interest for CPUs, but it is questionable whether they can be achieved with local memory.

We run the LMCOPY benchmark to measure the difference between global and local memory on CPUs. Kernel "global" copies data from a source matrix A to a target matrix B. Kernel "local" first loads A to local memory, followed by a barrier function. The barrier function performs as a synchronization point to ensure correct ordering of memory operations to local memory. Then it copies data from local memory to B. Both A and B reside in global memory.

In the Intel version, "local" generates 30% overhead for a 4K$\times$4K matrix, and it performs even worse in the AMD version. To verify the reason behind this behavior, we check the resulting assembly code and find out that new instructions and barriers are needed for copying and consistency, respectively. They both reduce performance.

We further analyze the use of local memory for data sharing. In the LMSUM benchmark, each work-item of a work-group performs: (1) $\text{Sum}_{group} \leftarrow$ sum reduction of A in the group; (2) B $\leftarrow \text{sum}_{group}$ + its own data in A. In "local", A is first loaded to local memory. The results (Table III) are opposite: "local" outperforms "global" by 37% in the Intel version, but performs more than twice worse in the AMD version.

TABLE III
LMSUM EXECUTION TIME (MS) DIFFERENCES BETWEEN GLOBAL AND LOCAL MEMORY.

4K$\times$4K	Intel OCL SDK 1.5	AMD APP SDK 2.6
Kernel "global"	358.1	336.6
Kernel "local"	260.8	878.4

From the Intel assembly code, we find that using local memory still has the copying overhead. But for "local", the compiler loads the data elements from local memory to registers, and executes the add operation in registers. In "global", the compiler directly performs the operation with one operand in register and the other in memory. It seems that the execution of a register-register (R-R) add together with a memory load is faster than that of a register-memory (R-M) add. The performance gain compensates for the local memory overheads. The AMD compiler does not apply the extra register copy, and therefore the performance remains bad.

Because global and local memory are in the same CPU memory hierarchy, the use of local memory on CPUs adds unnecessary overheads. For data sharing by local memory, the Intel compiler may optimize accesses via registers, and the use of R-R operations might lead to performance gain. The AMD compiler does not apply such optimization. Therefore, it is safer, performance-wise, not to use local memory on CPUs. If an OpenCL program does not use local memory, programmers should not spend time implementing it in the GPU way; if local memory is already in use, a comparison against a global memory version should show which one performs better.

E. Parallelism granularity and tiling

OpenCL, as a fine-grained programming model, provides programmers with the possibility of tuning problem decomposition. To figure out how this parallelism granularity influences CPU performance, we choose two benchmarks from Rodinia: PathFinder (PF) and Needleman-Wunsch (NW).

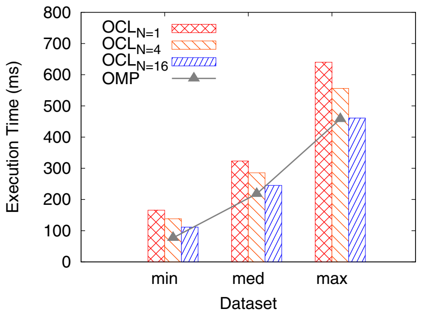
The core of the PF kernel compares a data element with its left and right neighbors, and chooses the minimum. The kernel is iterated row by row. We change the parallelism granularity by making each work-item process multiple adjacent data elements, and reduce the total number of work-items accordingly. Listing 2 shows the change of the core kernel part. Note that we use a loop to replicate the statement N times.

Listing 2. Increasing workload granularity per work-item

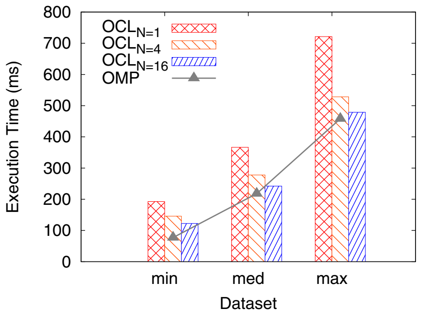
```
//before increasing workload granularity
__kernel void PF_before(__global int* matrix, int rid, int rows, int cols){
        int tid = get_global_id(0);
        int min = matrix[rid * cols + tid];
        int left = matrix[rid * cols + tid − 1];
        int right = matrix[rid * cols + tid + 1];
        min = MIN(min, left, right);

}

//after increasing workload granularity by N
__kernel void PF_after(__global int* matrix, int rid, int rows, int cols, int N){
        int tid = get_global_id(0) * N;
        for(int i = 0; i < N; i++){
                int min_i = matrix[rid * cols + tid + i];
                int left_i = matrix[rid * cols + tid + i − 1];
                int right_i = matrix[rid * cols + tid + i + 1];
                min_i = MIN(min_i, left_i, right_i);
        }
}
```

By increasing the workload granularity, we increase the data locality within each work-item and improve cache utilization. In Fig. 5, the OpenCL performance improves with the increase of N, and shows similar numbers to OpenMP for the larger N. The final performance gain (N=16) is 1.4× and 1.5× in the Intel and AMD versions, respectively.

In NW, each work-item performs a comparison among left, up, and diagonal neighbors, and the comparison is processed in diagonal direction in each iteration. As data access is not in a regular row/column-based order, the kernel uses tiling to decompose the problem space. Each work-group takes charge of one tile, and processes data elements diagonally. The tiling is done iteratively and also diagonally (Fig. 6).

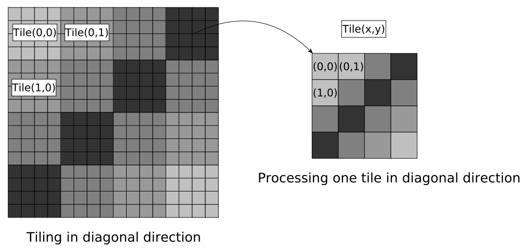


(a) Intel OCL SDK 1.5



(b) AMD APP SDK 2.6

Fig. 5. Execution time (ms) comparison of PF among different workload granularity. N=x represents increasing granularity by x times. PF uses datasets of 1600K, 3200K, 6400K data elements.



Tile(0,0)
Tile(0,1)
Tile(1,0)
Tile(x,y)
(0,0)
(0,1)
(1,0)
Processing one tile in diagonal direction
Tiling in diagonal direction

Fig. 6. Tiling in NW. In the whole matrix, Tile(0,0) is processed in the first outer iteration; Tile(1,0) and Tile(0,1) are processed concurrently in the second outer iteration. In each tile, Element(0,0) is processed in the first inner iteration; Element(1,0) and Element(0,1) are processed concurrently in the second inner iteration.

First, we investigate how the tile size influences performance. We assume that the time of processing each tile T_{tile} depends on the tile dimension D_{tile} (Eq.(1)). For a matrix with dimension D_{matrix}, the size of the outer loop L is determined by Eq.(2) (see Fig. 6). According to Eq.(3), the total execution time T_{matrix} should be determined by the tile size.

$$T_{tile} = f(D_{tile}), \tag{1}$$
$$L = D_{matrix}/D_{tile} \times 2 - 1, \tag{2}$$
$$T_{matrix} = T_{tile} \times L = f(D_{tile}) \times (D_{matrix}/D_{tile} \times 2 - 1) \tag{3}$$

Fig. 7 shows that the tile size has indeed visible impact on NW performance, and different compilers generate different performance trends. The speedup between tiling with 16×16 tiles and 1×1 tiles (degrade to no tiling) is 10.1× and 5.8× in the Intel and AMD versions, respectively. Therefore, when using tiling for OpenCL on CPUs, we need to choose an optimal tile size through experiments.

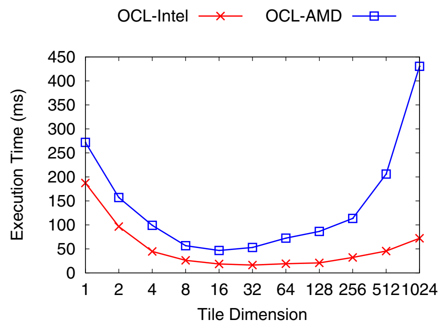


Fig. 7. Measured execution time (ms) of NW with tile dimension from 1 to 1024. The matrix size is 2K×2K. The optimal tile dimension is 16 – 32.

We further consider tiling in both OpenCL and OpenMP. As NW shows, tiling can be easily implemented in OpenCL with different tile sizes. But in OpenMP, this cannot be easily achieved by using compiler directives, and it depends on the ability of programmers to figure out a correct data layout rearrangement and complicated array index calculation. We use the performance model (Eq.(1-3)) to predict the ideal OpenMP performance with tiling. The time of processing one 16×16 tile is approximated by measuring the execution time for processing a 16×16 matrix, which is 0.24 ms. The predicted time for processing a 2K×2K matrix is 61.2 ms, which is larger than the original implementation with no tiling (24.5 ms). Eq.(4) summarizes the NW findings.

$$T^{Intel}_{OCL\_tiling} < T_{OMP\_orig} < T^{AMD}_{OCL\_tiling} < T^{Predicted}_{OMP\_tiling} \quad (4)$$

Overall, programmers should be aware that OpenCL's fine-grained parallelism can be both a disadvantage (PF) and an advantage (NW) for CPUs. The problem decomposition and the decomposition granularity should be (auto)tuned to make sure that CPU characteristics are properly addressed.

F. Explicit and implicit vectorization

CPUs provide SIMD units to speed up performance. Thus, programmers are encouraged to vectorize OpenCL code, and benefit from data level parallelism, but whether the way they use vectorization is appropriate needs to be examined.

One option is explicit vectorization, when OpenCL vector types (i.e., `int4`, `float4`) are used in the kernel. Compilers translate the vectorized code with SIMD instructions, and enable the use of SIMD units [17]. We use `int4` and `int16` to vectorize the PF kernel, and the speedup is around 1.2× and 1.8×, respectively. The method of explicit vectorization with `int4` is presented in Listing 3. Note that the number of work-items is reduced by 4, and vector data load is used.

Listing 3. Explicit vectorization using vector types

```
//explicit vectorization in PF
__kernel void PF_vecorization(__global int* matrix, int rid, int rows, int cols){
        int tid = get_global_id(0) * 4;
        //invoke vector data load function
        int4 min = vload4(0, &(matrix[rid * cols + tid]));
        int4 left = vload4(0, &(matrix[rid * cols + tid − 1]));
        int4 right = vload4(0, &(matrix[rid * cols + tid + 1]));
        min = MIN4(min, left, right);
}
```

Another alternative is implicit vectorization (only supported by the Intel compiler at present): the compiler transforms OpenCL scalar code into vectorized code by packing work-items together to run in SIMD fashion. The number of work-items packed is determined by the SIMD width[3]. In PF, the speedup of using implicit vectorization is 1.1×, a little lower than that of explicit vectorization (compared with Intel-`int4`), but the scalar code requires no changes.

TABLE IV
EXECUTION TIME (MS) AND SPEEDUP OF USING IMPLICIT VECTORIZATION IN CFD.

Dataset	Speedup	Execution time
fvcorr.domn.193K	1.6	42438.0
missile.domn.0.2M	1.8	80339.0

CFD from Rodinia also benefits from implicit vectorization. Table IV shows that CFD gets more performance improvement than PF through implicit vectorization. Comparing the execution time vertically, we notice that the execution time difference between the two datasets with similar size is quite large. This is because CFD contains data-dependent branches. In order to execute both the `if` and `else` parts, the `blend` instruction [18] is used as a mask to select SIMD lanes (as seen in the assembly code), and therefore the SIMD unit utilization is decreased. The two datasets have different branch penalties (they come from very different objects), which finally results in the large performance difference.

In summary, explicit and implicit vectorization lead to performance improvement. Explicit vectorization is more flexible, but its performance gain depends on the programmers' ability to optimize code. Implicit vectorization is easy to use, but the compiler's approach to vectorization may lead to performance penalties due to data-dependent branches.

G. Work-group size

Work-group size is the number of work-items within a work-group. Increasing work-group size reduces the number of work-groups. As different work-groups can be scheduled to run concurrently on different hardware cores/threads, a larger number of work-groups provides more flexibility in scheduling, while the switching overhead is also increased. Therefore, the choice of work-group size is a trade-off. For most applications we have tested, there is no general relation between the overall performance and the work-group size. One exception is in NW (see Fig. 7), where performance varies

[3]The CPU platform we use has 4-way SIMD units, and therefore 4 work-items are packed together.

depending on the work-group size (tile size). In addition, the optimal work-group size for GPUs is always not the optimal one for CPUs. Therefore, for an application already has a non-default size for GPUs: if the size is application-related, it might have to stay in place for CPUs; if the size is hardware-related, it needs to be tuned, and can be auto-tuned systematically.

IV. Discussion

In this section, we discuss the impact of our findings on OpenCL. We provide a set of guidelines for improving the programming style in OpenCL for CPUs, but we also provide arguments for the future embedding of these guidelines in (semi-)automated tools and/or compilers.

Table V summarizes the eight performance traps we have identified. This summary includes the CPU-OpenCL platform mismatch that is the cause of each trap (see Section II), the generic transformation needed to alleviate it, the concrete code operations and their potential for automation, as well as the applications we have used for exemplification, together with the performance gain (quantified as the speedup achieved after the proposed transformation(s)).

We note that there are three main categories of generic transformations needed for OpenCL codes to perform better on CPUs: (1) removing unnecessary copying, (2) improving data locality and cache utilization, and (3) enabling/improving the usage of SIMD units. Among these three categories, improving cache utilization generates the highest performance improvement, followed by the use of SIMD units and, the most obvious of the three, the removal of the unnecessary copying. Furthermore, we note that the specific code transformations we are recommending are interesting not by their novelty as optimization techniques, but rather by their generic applicability to multiple applications.

In an attempt to evaluate the impact of these CPU-friendly code transformations on the performance portability, we also include an estimate of the impact they have on the GPU OpenCL version. We note that the effects are mixed: vectorization is likely to improve GPU performance; granularity, tiling, and the work-group size will require GPU-specific tuning (the CPU parameters will not be the best for the GPU version); local memory removal and row-major memory access patterns will both incur GPU performance penalties. In other words, for current tools, performance portability will require some degree of code specialization. However, this specialization comes in the form of parameter tuning (e.g., changing granularity, switching between row- and column-major order, enabling/disabling data copying), and does not alter the application structure and parallelization.

Finally, Table V also shows how the CPU-friendly OpenCL code performs when compared with the reference OpenMP code. We note that the OpenCL code is on par or slightly better than the OpenMP code, an important indication that the proposed transformations do enable OpenCL to make good use of the CPU platform, at least as good as an average, non-aggressively optimized OpenMP version. We consider this as an important argument in favor of using already available OpenCL codes as a first, and virtually effort-free, performance estimate of a parallel version of a given application.

Given all these elements, we propose a set of guidelines, to be applied in order when porting/writing OpenCL CPU code:

1) Check the data transfer method and apply the zero copy technique.
2) Determine the problem decomposition granularity, and increase workload per work-item and/or apply tiling.
3) If the application processes 2D datasets, check the memory access pattern, and make sure row-major order is used in all places.
4) Apply explicit vectorization or try to enable implicit vectorization. When using implicit vectorization, data-dependent branches should be removed, if possible, to reduce the penalty.
5) Check experimentally whether the work-group size is optimal for maximizing performance.
6) Check the use of local memory, as the benefit of using local memory is not promised. Generally, local memory is not recommended on CPUs. For applications that already make use of it, it is better to compare with a non-local memory version, and pick the better one.

To summarize, we believe that achieving performance portability is a three-step process: identify problems, devise systematic solutions, and implement them into tools/compilers. Our work has addressed the first two steps, and has brought evidence that tools for (semi-)automated transformation of OpenCL GPU code into a CPU-friendly one are feasible. Our solutions might not achieve the best performance of a parallel implementation of a given application on CPUs, but they do provide an easy way to ensure acceptable performance. They also improve the flexibility in OpenCL's usage, which in turn should further boost the adoption of OpenCL as a parallel programming model for many cores.

V. Related Work

In this section, we present a brief overview of previous work on OpenCL performance on CPUs. There is much work on optimizing OpenCL for GPUs [19]–[21] and heterogeneous computing [22], [23], which shows that OpenCL is an interesting and promising programming model.

However very little research is available in analyzing OpenCL on many-core CPUs. Nieuwpoort et al. [24] evaluate a streaming, real-time application on: CPUs, NVIDIA and AMD GPUs, and the Cell/B.E. (Cell Broadband Engine). Their study points out that when dealing with OpenCL code on CPUs, programmers still have to consider architectural details to reach the best performance. Correspondingly, a suite of microbenchmarks, uCLbench, is proposed in [25] to characterize different hardware platforms (three CPUs, four GPUs, and one accelerator) to guide application algorithm design and optimization. While we share some of their end conclusions, our work presents a more thorough analysis focused entirely on the OpenCL performance issues on CPUs.

Intel and AMD both support OpenCL on CPUs with their own OpenCL drivers and compilers. They have also released programming guides [26], [27] aiming to teach programmers how to get correct performance on CPUs. Compared to these manuals, our work addresses a larger problem, as we not only provide "best practices", but we also identify the causes of performance problems, and we quantify their impacts.

TABLE V
PERFORMANCE TRAPS SUMMARY.

Trap	Mismatch	Generic transformation	Concrete code ops	Auto	Application	Speedup	GPU	OMP
H2D+D2H	memory model	avoid copying	zero copy	Y	HDCOPY	2.5 - 6.0	×	≃
Memory access pattern	memory model	cache utilization	row-major access	Y	SC, KM	4.0 - 10.0	↓	≥
Local memory	memory model	-	remove	-	LMCOPY/SUM	-	↓	≥
Workload granularity	granularity	cache utilization	increase granularity	Y	PF	1.5	↑↓	≃
Tiling	granularity	cache utilization	tiling (tune size)	Y	NW	5.8 - 10.1	↑↓	≥
Explicit vectorization	granularity	SIMDzation	use vector type	Y/N	PF	1.2 - 1.8	↑	≃
Implicit vectorization	granularity	SIMDzation	enable	Y	PF, CFD	1.1 - 1.8	×	≃
Work-group size	granularity	HW-related	tuning	Y	*all*	HW-related	↑↓	≥

Performance on GPUs would increase (↑), decrease (↓), depend on tuning (↑↓), not applicable (×).
OpenCL performs no less than (≥) or similar (≃) with OpenMP after code transformation.

VI. CONCLUSION AND FUTURE WORK

OpenCL is a cross-platform parallel programming model, which should be used on GPUs and CPUs alike. While the model does guarantee (to a large extent) code portability, performance portability is still to be achieved. In this paper, we have analyzed the performance of OpenCL code from the perspective of portability. Specifically, we have focused on one question: how does OpenCL code perform on CPUs?

We started our analysis from the observations that OpenCL (heavily inspired by the GPU architectures and the CUDA model) (1) does not match the CPU architecture in parallelism granularity and (2) has a different memory model. These mismatches have lead to the identification of eight performance traps, leading in turn to significant performance degradation. Alleviating the effects of these traps requires systematic solutions to transform the original GPU-inspired OpenCL implementations into a CPU-friendly form, and can bring performance improvements up to 10×. These transformations target three main aspects: (1) data locality and cache utilization, (2) unnecessary copying removal, and (3) SIMD unit utilization. Our results show that increasing cache utilization generates the best improvement for CPUs. Furthermore, these CPU-friendly implementations are comparable, performance-wise, with OpenMP reference implementations.

The CPU-friendly form is, in fact, a specialization of the code, which now has a CPU and a GPU form. However, the transformations we propose do not alter the parallelization or the structure of the original OpenCL code. For now, our work helps both programmers and compiler developers to further improve OpenCL performance on CPUs. In the near future, we aim to implement these transformations as a parameterization of the OpenCL code. In this case, code specialization becomes a generic form of parameter tuning, providing good opportunities for automation and a bigger step in the direction of inter-platform OpenCL performance portability.

REFERENCES

[1] K. Group., "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems," 2012. http://www.khronos.org/opencl/.

[2] NVIDIA, "CUDA C Programming Guide." http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[3] K. Group., "The OpenCL Specification V1.0," 2009. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

[4] A. Ali, U. Dastgeer, and C. Kessler, "OpenCL for Programming Shared Memory Multicore CPUs," in *MULTIPROG, in conjunction with HiPEAC 2012*, 2012.

[5] K. Thouti and S. Sathe, "Comparison of OpenMP & OpenCL Parallel Processing Technologies," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 4, 2012.

[6] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W.-m. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[7] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," in *GPGPU 2010*, pp. 63–74, ACM, 2010.

[8] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE IISWC2009*, pp. 44–54, 2009.

[9] E. Ayguade, N. Copty, A. Duran, *et al.*, "The Design of OpenMP Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, 2009.

[10] D. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[11] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[12] I. Inc., "Intel Array Building Blocks (Intel ArBB)." http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[13] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, "Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data," in *ISCAW 2011 (A4MMC)*, 2011.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[15] K. Group., "The OpenCL Specification V1.2 Document Revision 15," 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[16] Intel, "Intel Hyper-Threading Technology." http://www.intel.com/, 2012.

[17] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," in *Compiler Construction 2012*, pp. 1–20, Springer, 2012.

[18] Intel, "Intel 64 and IA-32 Architectures Software Developer Manual." http://www.intel.com/, 2012.

[19] 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 Computing*, 2011.

[20] M. Daga, T. Scogland, and W. Feng, "Architecture-Aware Mapping and Optimization on a 1600-Core GPU," in *IEEE ICPADS 2011*, pp. 316–323, 2011.

[21] K. Komatsu, K. Sato, Y. Arai, *et al.*, "Evaluating Performance and Portability of OpenCL Programs," in *iWAPT 2010*, 2010.

[22] J. Lee, J. Kim, S. Seo, *et al.*, "An OpenCL Framework for Heterogeneous Multicores with Local Memory," in *PACT 2010*, pp. 193–204, ACM, 2010.

[23] J. Breitbart and C. Fohry, "Analyzing Use of OpenCL on the Cell Broadband Engine and a Proposal for OpenCL Extensions," *IJNC*, vol. 1, no. 1, pp. 114–130, 2011.

[24] R. Nieuwpoort and J. Romein, "Correlating Radio Astronomy Signals with Many-Core Hardware," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 88–114, 2011.

[25] P. Thoman, K. Kofler, *et al.*, "Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design," in *Euro-Par 2011*, pp. 438–452, Springer, 2011.

[26] Intel, "Intel SDK for OpenCL Applications 2012 Optimization Guide." http://software.intel.com/sites/landingpage/opencl/optimization-guide/index.htm, 2012.

[27] AMD, "AMD Accelerated Parallel Processing OpenCL Programming Guide." http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf, 2012.