Delft University of Technology
Parallel and Distributed Systems Report Series

# CLVectorizer: A Source-to-Source Vectorizer for OpenCL Kernels

**Jianbin Fang, Ana Lucia Varbanescu**
{j.fang,a.l.varbanescu}@tudelft.nl

Completed in November 2011

**Report number PDS-2011-012**

TU Delft PDS

**ISSN 1387-2109**

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Group:
http://www.pds.ewi.tudelft.nl/



J. Fang, A.L. Varbanescu

Source-to-Source Vectorization for OpenCL Kernels

**Abstract**

While many-core processors offer multiple layers of hardware parallelism to boost performance, applications are lagging behind in exploiting them effectively. A typical example is vector parallelism(SIMD), offered by many processors, but used by too few applications.

In this paper we discuss two different strategies to enable the vectorization of naive OpenCL kernels. Further, we show how these solutions are successfully applied on four different applications on three different many-core platforms. Our results demonstrate significant improvements in both achieved bandwidth and execution time for most (application, platform) pairs. We conclude therefore that vectorization is not a computation-only optimization for OpenCL kernels, but one that enables the applications to better utilize the hardware.

Using our experience with the vectorization, we present a refinement of the process into a two-module framework to assist programmers to optimize OpenCL code by considering the specifics of the target architectures. We argue that such a framework can further speedup applications based on the current work, and we also show what are the requirements for making such an extension.

**Keywords:** SIMD, Vectorization, OpenCL.

# Contents

# List of Figures



# List of Tables

# 1 Introduction

Many-core processors are massively parallel architectures, designed and built to achieve performance by exploiting multiple layers of parallelism. Using languages like OpenCL and CUDA, many applications are being relatively easily ported onto many-cores, with spectacular results when compared with their original (typically sequential) versions. However, these applications are forming a large code base of naive implementations: functionally correct, but still far from the potential performance they could achieve if properly optimized.

There are several classes of generic optimizations that are applied for many-core applications [1]. In this paper, we focus on *code vectorization* in the context of SIMD (single instruction multiple data) cores [2]. By *vectorization* we understand the transformation of *scalar code* (i.e., code using scalar data types and instructions) into *vectorized/SIMD code* [3].

Currently, there are multiple many-core architectures that use SIMD cores. For example, AMD's GPUs [4] use hundreds of *stream* cores, each of which is an SIMD processor with four or five processing elements. Moreover, Intel has recently introduced AVX (Advanced Vector eXtension) [5], an extension of the previous SIMD instruction sets (SSE - streaming SIMD extensions) aimed at speeding up computationally intensive floating point applications. AVX is supported by the recently released Intel Sandy Bridge processor, and it is also to be supported by the AMD Bulldozer architecture. Note that, in fact, NVIDIA's GPUs, with their scalar processing elements, are more of an exception than a rule. For these architectures, vectorization is a mandatory optimization: without it, applications literally waste more than 50% of the processing power.

Vectorization is not a new concept - vector processors have been around for more than 30 years and various compiler-based attempts to automate code vectorization have proved partially successful [3]. The problem was never completely solved, but modern compilers are able to partially vectorize user code in order to use an SIMD instruction set. The most recent example is the Intel OpenCL SDK compiler [6], which has a vectorization module that takes scalar OpenCL code and rewrites it using SIMD data types and instructions. However, the transformation techniques these compilers use are not visible to the programmer, who is therefore unable to track and/or tune the changes. Furthermore, for complex codes, most compilers remain conservative and drop aggressive solutions.

In this paper, we discuss an alternative solution for code vectorization: instead of focusing on automated vectorization for generic cases, we propose two different vectorization approaches, specifically designed for naive (i.e., unoptimized) OpenCL code. Essentially, we are using a source-to-source translator that starts from a generic (scalar) kernel and applies step-by-step transformations to obtain a vectorized one.

We have tested our source-to-source vectorization on four different types of benchmarks - namely, Matrix Multiplication (MM), 2D-Image Convolution (IC), Black-Scholes (BS), and Red-Black SOR (SOR). We compare the performance results of the vectorized and non-vectorized codes on three different architectures: NVIDIA's GeForce GTX580, AMD's Radeon HD6970, and Intel's Xeon X5650. Furthermore, we investigate the effects of vectorization on both the bandwidth and the overall kernel execution time. We note interesting performance results: in most cases, vectorization boosts the application performance (improvements ranging between 4% and 300%), but can also slow-down certain applications (up to 11%). Finally, as we believe this is a promising approach for OpenCL kernels optimization, we propose a framework to refine this solution and further extend it by considering the architecture specifics.

Our main contributions are as follows.

- We discuss two orthogonal kinds of source-to-source vectorization, intra-vectorization and inter-vectorization and their implementations (see Section 2); further, we analyze their impact on overall application performance (in Section 4).
- We present a detailed investigation on the effects of vectorization on several typical memory access patterns (MAP), and we show that both implicit data caching and explicit data reuse contribute to the bandwidth increase (see Section 3), which proves that vectorization also brings us a better utilization on memory access, not only computational power.

- We extend our vectorization approach into the CLVectorizer framework (in Section 5), and we show how it can be integrated with the current work and possibly further improve performance.

Based on our hands-on experience with vectorization, we believe we can generalize the optimization process of naive OpenCL code to improve its performance portability. Therefore, our conclusions (see Section 7) are two-fold: (1) code vectorization improves platform utilization and (2) OpenCL source-to-source vectorization can be generalized as a platform-agnostic optimization.

# 2 Source-to-Source Vectorization

In this section we briefly discuss OpenCL (our target language), and we focus on presenting a solution for OpenCL code vectorization. Specifically, we discuss two different types of vectorizations suitable for OpenCL kernels, and we show what code transformations have to be applied to enable them.

## 2.1 OpenCL as an Intermediate Language (IL)

OpenCL (Open Computing Language) has emerged as an open standard for programming many-cores, defined and managed by Khronos Group [7]. Designed as a cross-platform environment, it has been adopted by Intel, AMD, NVIDIA, IBM, and ARM. Due to its flexibility and portability, the model is currently a viable solution for building applications that execute across heterogeneous platforms consisting of hosts - general purpose CPUs, and multiple devices - typically accelerators like GPUs and FPGAs, but also many-core CPUs. Note that the memory spaces of the host and the device are (logically) separated.

In OpenCL terms, applications are composed of "host code" - i.e., the code that is meant to run on the hosts, and "kernel code" - i.e., the massively parallel code that runs on the devices. OpenCL includes a special language (based on C99) for writing kernels and the APIs that are used to define and then control the platforms. Several OpenCL bindings are available to higher level languages (Java, Python, C++), allowing to write host code in all these languages.

When a kernel is submitted for execution by the host, an index space, *NDRange*, is defined. An instance of the kernel is known as a *work-item*. Work-items are organized into *work-groups*, providing a more coarse-grained decomposition. Each work-item has its own private memory space, and can share data via local memory with the other work-items in the same work-group. All work-items can read/write global device memory.

In terms of parallelism, OpenCL provides high-level task- and data-parallelism. It also supports SIMD parallelism by providing vector data types as `char`*n*, `int`*n*, and `float`*n*, and ways to access their components. It also extends operators (e.g., arithmetic operators, relational operators, logical operators, etc.) based on element-wise operations. Finally, OpenCL C has build-in functions that support vector operations, such as cross product and dot product [7].

Focusing on vectorizing kernels, we use OpenCL as both the input and the output language. We argue that OpenCL is a good choice of an intermediate language due to its cross-platform feature, which allows the vectorization to be applied for multiple platforms.

## 2.2 Alternatives for vectorization

Traditionally, vectorization is loop-based: once a loop is determined to be vectorizable, the loop is strip-mined by vector length and each scalar instruction is replaced by the corresponding vector instruction (i.e., the operands are vectors, and the operation is element-wise) [8]. In other words, traditional vectorization is an *aggregation of consecutive loop iterations*.

When it comes to many-core programming in OpenCL, the basic unit of computation is *work-item*. In this case, we propose to explore two different types of vectorization: *inter-vectorization (eV)*, and *intra-vectorization*

Listing 1: native kernel

```
#define N 32

__kernel void native(const __global float * in, __global float * out){

        int idx = get_global_id(0);
        float val = 0.0;
        for(int i=0; i<N; i++){
          val += in[idx+i];
        }
        out[idx] = val;
}
```

*(iV)*. Both strategies focus on kernels, and only $eV$ requires a minor change (i.e., an adjustment to the overall number of work-items) in the host programs.

For *inter-vectorization (eV)*, we enable vectorization across work-items. $eV$ is based on work-items merging, a technique to merge multiple neighboring work-items as a new work-item with coarse granularity, reducing the number of work-items by $VF$ (vectorization factor). This technique is used by Yi Yang et. al. to enhance data sharing [1]. In our work, we merge work-items, and store operands using vector data (the same size with the physical SIMD register). In theory, $eV$ can be applied to any OpenCL kernel code. Furthermore, $eV$ requires a minor change in the host program, to reduce the number of work-items by $VF$.

In *intra-vectorization(iV)*, we focus on vectorizing the work performed per work-item. $iV$ is based on loop unrolling (much like the traditional vectorization). Specifically, we unroll the loop for $VF$ times, and rewrite the scalar computation into its vector form. There is still work to be done to deal with the loop remainder when the loop boundary is not multiple of $VF$. Note that $iV$ can only applied to kernels with loops.

We note that the two approaches - $iV$ and $eV$ - are orthogonal, but not exclusive. In other words, they can both be applied individually or together on the same kernel, with different performance results (see Section 4 for a detailed analysis of several performance results).

## 2.3 Code transformations for vectorization

Common practice shows that OpenCL kernels have four parts: (1) work-item index calculation, (2) loading data from device memory, (3) computation, and (4) writing data back to device memory. In the following paragraphs, we describe the transformations to be made to each of these parts when enabling vectorization.

To make clear how both $eV$ and $iV$ work, we will be using a running example, shown in Listing 1.

### 2.3.1 Applying $eV$

We propose a two-step translation of an OpenCL kernel to its vectorized format using inter-work-item vectorization.

**Step 1-** Duplication:

- Preserve: the kernel function definition (function name, arguments and data types), and control flow statements (e.g., `if` or `for` statements).
- Re-calculate work-item index (i.e., times $VF$ in the x direction).
- Duplicate declarations and expressions for $VF$ times.
- Re-name variables with unique identifiers, still derived from the original name - e.g., use a 'name_counter' scheme.

Listing 2: $eV$: step 1

```
#define N 32
#define VF 4

__kernel void inter_1(const __global float * in, __global float * out){

        int idx = get_global_id(0) * VF;
        float val_0 = 0.0;
        float val_1 = 0.0;
        float val_2 = 0.0;
        float val_3 = 0.0;
        for(int i=0; i<N; i++){
                val_0 += in[idx+i+0];
                val_1 += in[idx+i+1];
                val_2 += in[idx+i+2];
                val_3 += in[idx+i+3];
        }
        out[idx+0] = val_0;
        out[idx+1] = val_1;
        out[idx+2] = val_2;
        out[idx+3] = val_3;
}
```

Listing 3: $eV$: step 2

```
#define N 32
#define VF 4

__kernel void inter_2(const __global float * in, __global float * out){

        int idx = get_global_id(0) * VF;
        float4 val = (float4)0.0;
        for(int i=0; i<N; i++){
                val += (float4)(in[idx+i+0], in[idx+i+1], \
                        in[idx+i+2], in[idx+i+3]);
        }
        __global float4 * view_o = (__global float4 *)out;
        out[idx] = val;
}
```

- Recognize data sharing - i.e., check data reading statements to see whether there is overlapped data among the original $VF$ work-items.
- Duplicate writing-back statements.

**Step 2-** Vectorization:

- Replace scalar data with vector data. Note that when using vector data type for writing operations, one can take advantage of write buffer (if there is any available on the target platform [4]) .

The vectorized forms of the original kernel (Listing 1) are shown in Listing 2 and Listing 3.

### 2.3.2 Applying $iV$

Here we show the rules of translating an OpenCL kernel to its vectorized format using loop unrolling.

**Step 1-** Unrolling:

- Preserve the kernel function definition (function name, arguments and data types), and the index calculation.
- Unroll the *for* loop $VF$ times (e.g., $VF$=4).
- Apply reduction for the calculation of the final result (e.g., a sum).
- Preserve the data writing part.

**Step 2-** Vectorization:

- Replace the scalar data with vector data.

Note that we do not need to make any changes on the host program. The transformations of the naive kernel (Listing 1) based on these rules are shown in Listing 4 and Listing 5

Listing 4: $iV$: step 1

```
#define N 32

__kernel void intra_1(const __global float * in, __global float * out){

        int idx = get_global_id(0);
        float val_0 = 0.0;
        float val_1 = 0.0;
        float val_2 = 0.0;
        float val_3 = 0.0;
        for(int i=0; i<N; i=i+4){
                val_0 += in[idx+i+0];
                val_1 += in[idx+i+1];
                val_2 += in[idx+i+2];
                val_3 += in[idx+i+3];
        }

        out[idx] = val_0 + val_1 + val_2 + val_3;
}
```

Listing 5: $iV$: step 2

```
#define N 32

__kernel void intra_2(const __global float * in, __global float * out){

        int idx = get_global_id(0);
        float4 val = (float4)(0.0);

        for(int i=0; i<N; i=i+4){
                val += (float4)(in[idx+i+0], in[idx+i+1], \
                        in[idx+i+2], in[idx+i+3]);
        }

        out[idx] = val.x + val.y + val.z + val.w;
}
```

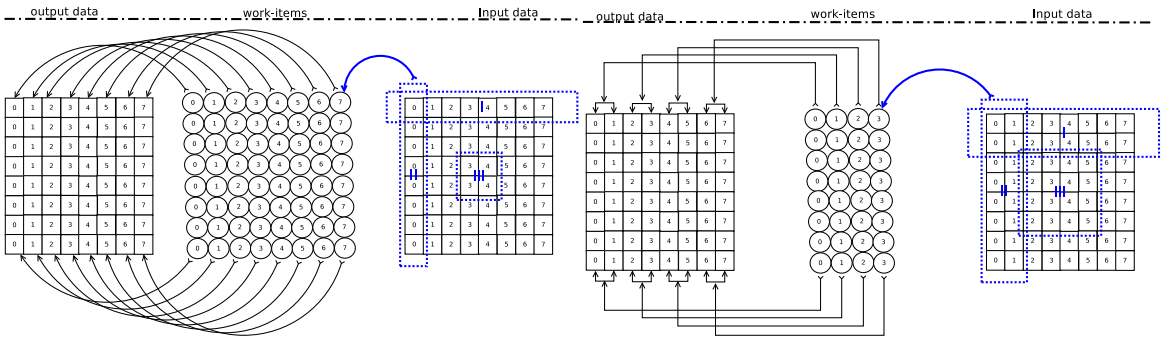
.

# 3 Vectorization Effects on Memory Bandwidth

In this section, we discuss the effects of vectorization on memory bandwidth by using several applications with different MAPs. We explain typically used MAPs at two levels: work-item (WI) level and work-group (WG) level.

For data parallel applications, we usually decompose problems based on output data, shown in Figure 1 (the circle array represents a grid of work-items, and the square array represents input/output data). Each work-item works on one element in the output data, and needs to read the corresponding elements from the input data ( (I) one row, (II) one column, (III) one block of elements, or multiple elements in random formats, shown in Figure 1a). On the WG level, neighboring work-items access the same or separate elements simultaneously, also presenting access patterns. For simple and clear demonstration, we suppose there are 8 work-items per work-group.

When introducing $eV$, MAPs change, i.e., each work-item works on $VF$ elements of output data, and its input data needed is expanded, as shown in Figure 1b ($VF$=2). In this section, we discuss the effects of this change on the memory bandwidth on AMD HD6970, Intel X5650, and NVIDIA GTX580.



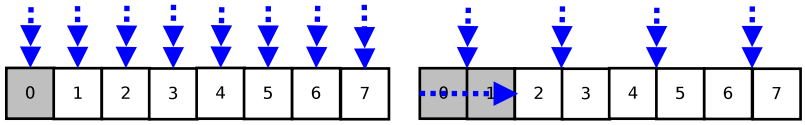

(a) Without vectorization (b) With vectorization

Figure 1: Mapping work-items to data. Circles represent a grid of work-items, squares represent input/output data arrays. There are 64 (8x8) work-items altogether, and each work-item computes one element from the output data. When using $eV$, each work-item works on two output elements.

Note that in the description of MAPs, we consider the mapping between work-items and output data to be 1:1, or 1:*VF*. Therefore, we do not include the output data and the work-item grid in our figures for clarity. The shaded area covers the elements accessed by one work-item. The dashed line with single arrow represents the memory access order by one work-item, and the dashed line with double arrows represents how (8) neighboring work-items access input data.

### 3.0.3 MAP SS

Each work-item reads one element from the input data, and writes the result back when finishing computation (see Figure 2). Neighboring work-items access separate elements. When using *eV*, each work-item will access *VF* physically continuous data elements; the distance between neighboring work-items becomes *VF*, rather than 1.




(a) Without vectorization (b) With vectorization

Figure 2: MAP SS (*VF=2*)

Figure 3 shows we can achieve bandwidth improvement of 1.26x, 1.39x, 1.60x on HD6970, X5650, GTX580, respectively. The bandwidth improvements result from read-data cache, or write-data cache, or both: HD6970 keeps separate read and write caches, while they work as one entity in GTX580 and X5650. This MAP is found in applications like Black Scholes.

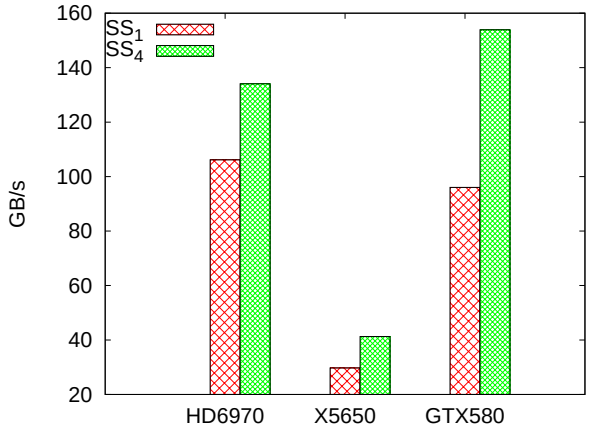
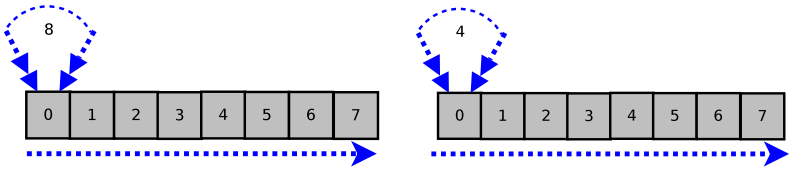



Figure 3: Bandwidth comparison when using MAP SS (*VF=4*)

### 3.0.4 MAP RF

Each work-item accesses a whole row of elements (see Figure 4). From the WG-level perspective, neighboring work-items will access the same data element each time. When using *eV*, the number of work-items is reduced by *VF* times, and each work-item will access the same element for *VF* times, leading to register data sharing. Therefore, we can achieve bandwidth improvements of 3.95x, 3.00x, and 2.63x when using *eV* on HD6970, X5650,

and GTX580 respectively, as shown in Figure 5. This MAP is a common one found in many applications, such as Matrix Multiplication.




(a) Without vectorization (b) With vectorization

Figure 4: MAP RF (*VF=2*)

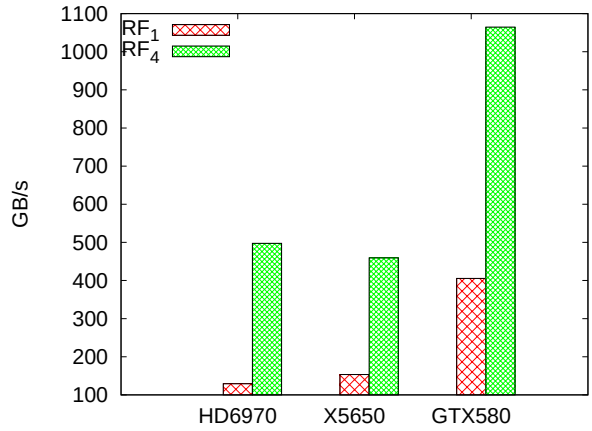



Figure 5: Bandwidth comparison when using RF (*VF=4*)

### 3.0.5 MAP CS

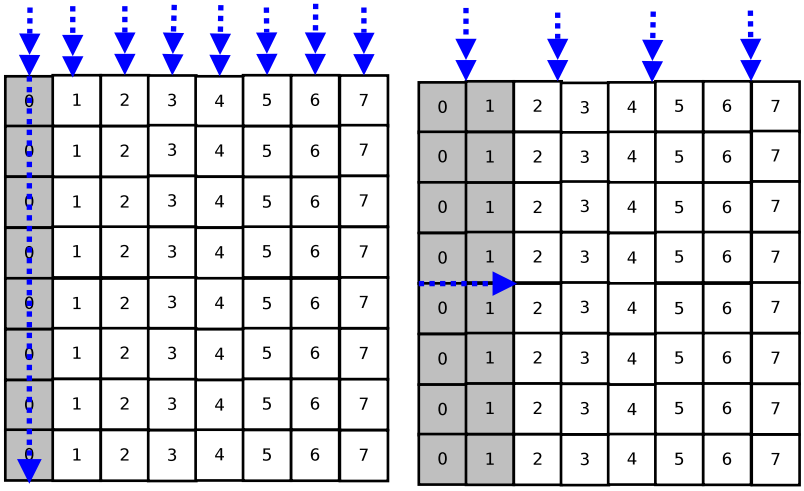
Each work-item loads a whole column of data elements (see Figure 6). On the WG-level, neighboring work-items will access spatially close data elements. When using $eV$, the number of work-items is also reduced by $VF$ times. Each work-item will access $VF$ columns of data elements, and the distance with its neighboring work-items is $VF$, rather than 1.

From Figure 7, we can see that the bandwidths with vectorization are 3.31x, and 1.76x higher than without it on HD6970 and GTX580. For X5650, the results are different: we obtain similar bandwidths when using $eV$. This MAP is found in applications like Matrix Multiplication.

### 3.0.6 MAP BS

Each work-item will read a block of data elements from the input data (see Figure 8). Neighboring data will access spatially closed data elements. When using $eV$, the data block used by a single work-item is expanded (the size of expansion area depends on $VF$), which enables data sharing when computing different output data elements. The distance between neighboring threads is $VF$, rather than 1.

From Figure 9, we can see that the bandwidth is around 2.66x higher than the one without vectorization on HD6970, while on GTX580, there is little gain in bandwidth. For the X5650, the bandwidth with vectorization is much lower. After explicitly enabling data sharing using register shuffling in kernel code, we obtain an improved



(a) Without vectorization (b) With vectorization

Figure 6: MAP CS (*VF=2*)

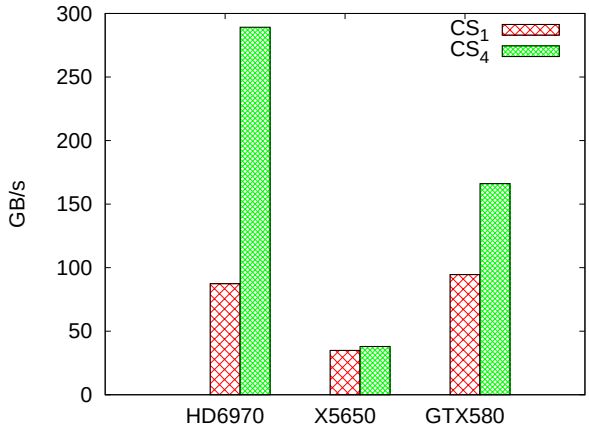


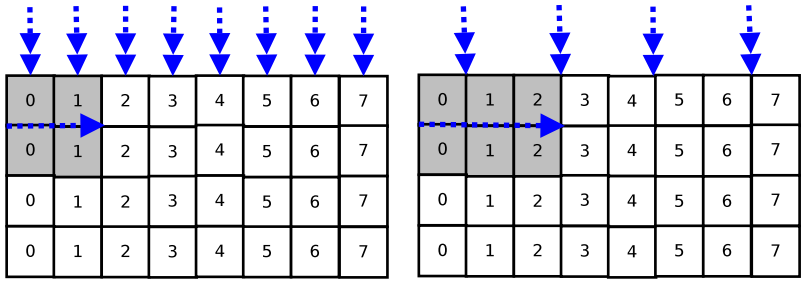

Figure 7: Bandwidth comparison when using MAP CS (*VF=4*)

bandwidth of 2.89x, 1.17x, and 1.86x on HD6970, X5650, and GTX580 respectively (shown in 9b, 9c, and 9d). This MAP appears in applications like 2D Image Convolution and Stereo Vision [9].

### 3.0.7 MAP NS

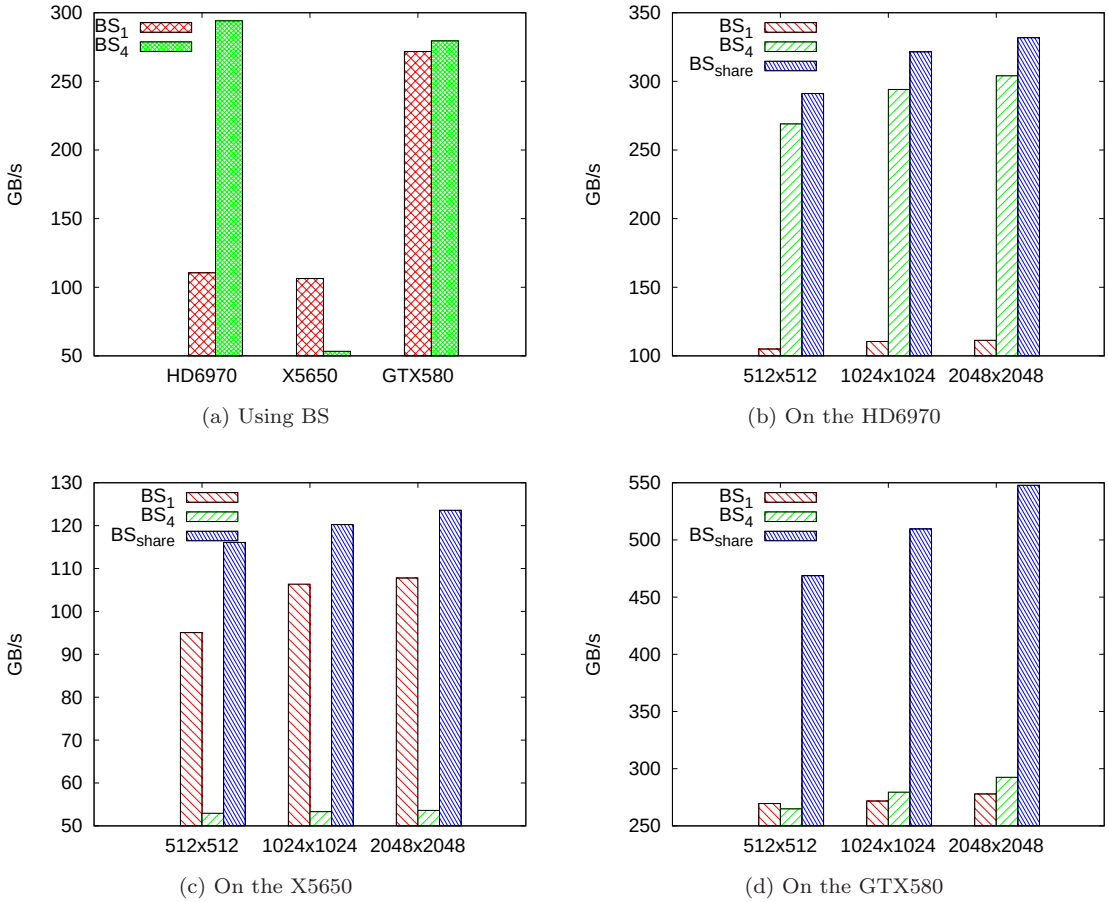
Each work-item will access the physically-closed four (left, right, top, bottom) elements around it (see Figure 10). Neighboring work-items access neighboring data elements. When using vectorization, the data elements accessed by one work-item will become *VF* times as many as before.

Our experimental results (see Figure 11) show that we can achieve 1.18x bandwidth improvement on HD6970. However, the bandwidth with vectorization is lower (by 4% and 19%) than the bandwidth without vectorization on X5650 and GTX580. MAP NS is similar to MAP SS due to the fact that there is no/little data sharing across work-items, and both can make use of the write combine buffer. However, MAP NS shows bandwidth




(a) Without vectorization (b) With vectorization

Figure 8: MAP BS (*VF=2*)



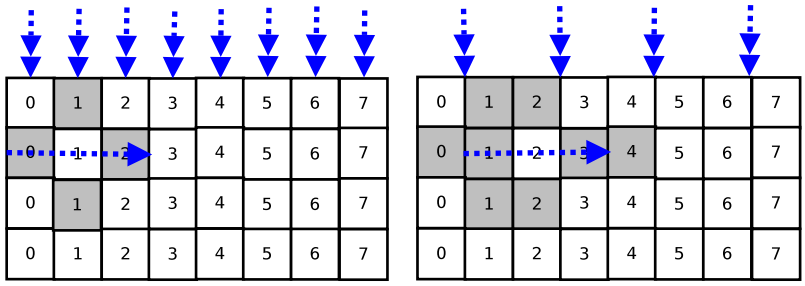

(a) Using BS (b) On the HD6970

(c) On the X5650 (d) On the GTX580

Figure 9: Bandwidth comparison when using BS and using data shuffling (*VF=4*)

degradation on X5650 and GTX580 possibly due to limited cache capacity. This MAP is found in applications such as Red-Black SOR.

To summarize, bandwidths with vectorization are higher than those without it for most (MAP, platform)




(a) Without vectorization (b) With vectorization

Figure 10: MAP NS (*VF=2*)

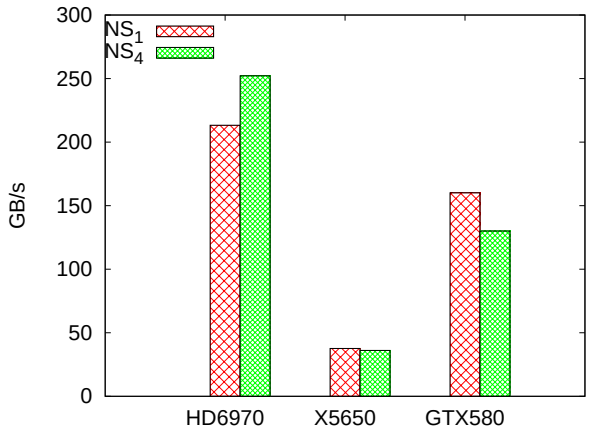



Figure 11: Bandwidth comparison when using MAP NS (*VF=4*)

pairs, as shown in Table 1. Two factors contribute to the bandwidth improvement: (1) implicit data reuse via caching; (2) explicit data sharing via register shuffling. However, we also see some performance degradation on bandwidth, either because the interference of compilers (i.e., compilers have made some other optimizations), or because of limited register/cache capacity.

# 4 Vectorization Effects on Overall Performance

In this section, we discuss the improvements that vectorization brings to the overall kernel performance.[1]

## 4.1 *Inter-vectorization (eV)*

First, we discuss how $eV$ affects the kernel execution time on the three selected platforms, shown in Figure 12. On HD6970, the vectorized kernels perform better than the naive kernels: the speedups are 3.27x, 3.03x, 1.26x, and 1.18x for MM, IC, BS and SOR, respectively. On X5650, the vectorized IC, BS and SOR perform 1.20x, 1.04x, and 1.44x faster than the naive kernel. MM with our vectorization performs slightly worse than the naive kernel. Note that the overall speedup is limited by the smaller bandwidth when there are two or more inputs.

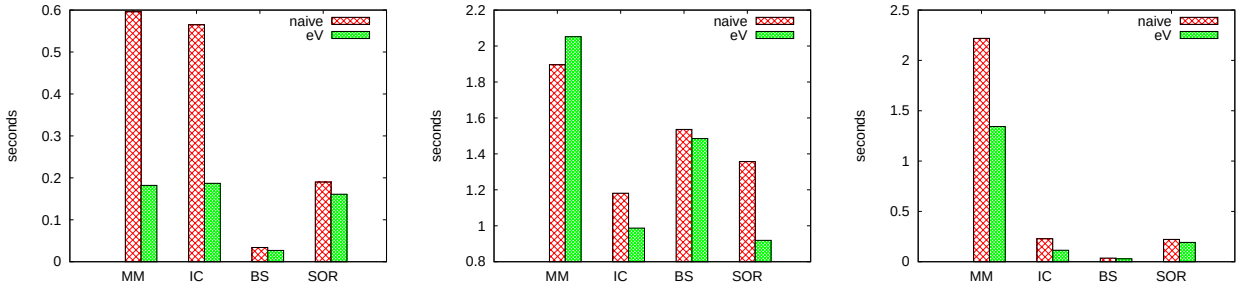
[1]The source code is available here: http://code.google.com/p/clvectorizer/.

Table 1: MAPs summary

| Name | WI Level | WG Level | Bandwidth Gain(x) | | | Applications |
|---|---|---|---|---|---|---|
| | | | HD6970 | E5620 | GTX580 | |
| SS | single | separate | 1.27 | 1.51 | 1.6 | Black Scholes |
| RF | row | focused | 3.85 | 2.92 | 2.62 | MM, Matrix A (A*B=C) |
| CS | column | separate | 3.33 | 0.97 | 1.74 | MM, Matrix B (A*B=C) |
| BS | block | separate | 2.9 | 1.13 | 1.86 | Image Convolution |
| NS | neighbors | separate | 1.18 | *0.96* | *0.81* | SOR |

For example, for matrix multiplication (A*B=C), the bandwidth improvement for matrix A and B are 3.85x and 3.31x; however, the overall speedup is 3.27x.

We select NVIDIA GTX580 as the scalar architecture to evaluate how the vectorized kernels perform. We can see that the vectorized kernels are improved by 1.65x, 2.00x, 1.26x, and 1.16x, compared with the naive ones for MM, IC, BS, and SOR. This is due to the bandwidth improvement introduced by vectorization, not to the usage of SIMD processing elements.




(a) On the GPU HD6970 (b) On the CPU X5650 (c) On the GPU GTX580

Figure 12: *eV* Performance on the three architectures

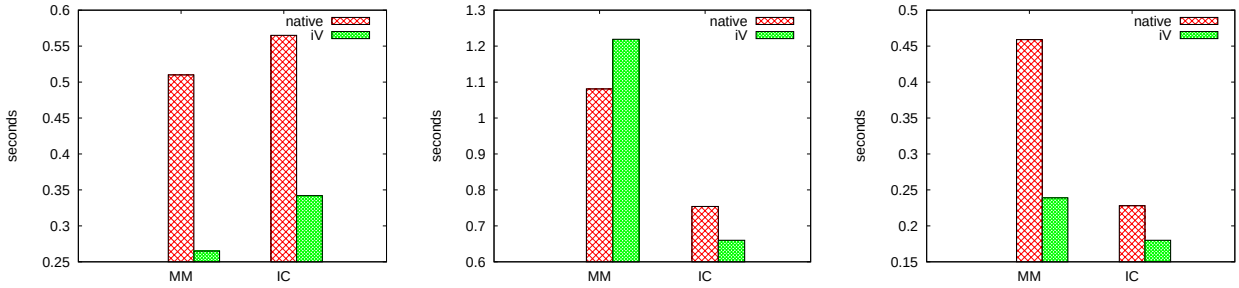
## 4.2 *Intra-vectorization (iV)*

For these kernels with loops, we can use *iV* by unrolling the loops *VF* times. The performance changes are shown in Figure 13, from which we can see that MM and IC perform better (1.93x, 1.65x on HD6970, and 1.92x, 1.27x on GTX580) than the naive versions. On X5650, the vectorized IC performs better than the original implementation (by 1.14x). However, the vectorized MM shows a performance loss of 11%. We believe this is a result of the compiler optimizations being more aggressive than the optimizations we perform.

# 5 The CLVectorizer framework

In this section, we present the CLVectorizer framework as an extension to the vectorization approach.

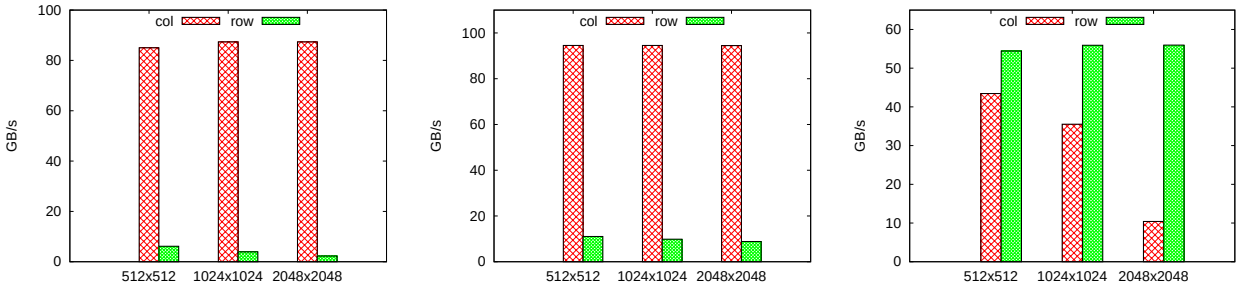
As shown in Section 3 , MAPs present two levels: work-item level and work-group level. GPUs pay more attention to work-group level to achieve coalescing access, i.e., neighboring work-items access physically closed data elements, while multi-core CPUs are focusing on the work-item level to exploit data locality via cache. A proof of this can be found in Figure 14, which shows that HD6970 and GTX580 can achieve much higher bandwidth when each work-item accessing memory in column-major order, and X5650 prefers the row-major




(a) On the GPU HD6970 (b) On the CPU X5650 (c) On the GPU GTX580

Figure 13: $iV$ Performance on the three architectures

pattern per work-item. Therefore, we should use the suitable MAPs for target architectures to maximize memory bandwidth.




(a) On HD6970 (b) On GTX580 (c) On X5650

Figure 14: Memory bandwidth when each work-item accesses memory in row-major order and column-major order on the three architectures.

Based on the previous vectorization work, our proposed CLVectorizer consists of two core modules: (1) MAPer, used to recognize memory access patterns presented in programs/applications, and to transform the input data to better match the MAP suitable for the target platforms, and (2) VECTORizer, which is used to perform the actual transformation/vectorization of the OpenCL kernel code (see Figure 15). With this approach, we can further improve the performance taking into account the specifics of the target architectures. The implementation of this framework is currently the work in progress.

# 6 Related Work

In this section, we give a brief overview of prior work on vectorization and optimizations based on memory access patterns.

With the advent of vector computers, there have been increased interests in making vector operations available. In [3], Allen and Kennedy present a translator to transform programs from FORTRAN to FORTRAN 8x. From around 2000, programmers have been employing multimedia extensions by using in-line assembly routines or specialized library calls to increase the performance. However, this error-prone process is exacerbated by inconsistencies among different instruction sets [8]. Thus, lots of vectorizing techniques have been proposed

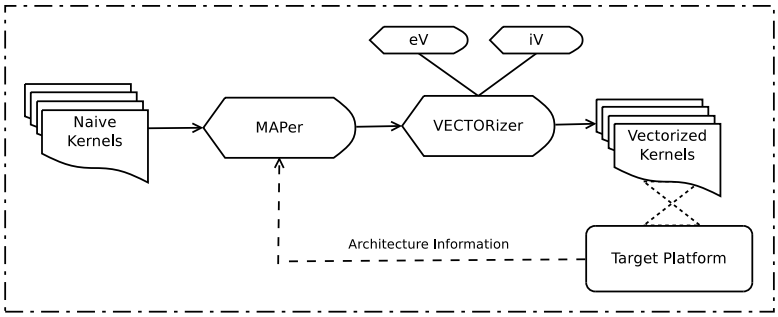



Figure 15: The CLVectorizer framework

either for loops [10], or for basic blocks [11]. These vectorizers can translate C programs to programs with inline assembly code. Our *intra-vectorization* is similar to the traditional vectorization, i.e., both are based on loop-unrolling/strip-mining. However, the *inter-vectorization* achieves vectorization through merging work-items. Our approach can also address the issue of portability due to the OpenCL-based implementation.

Memory access patterns have been explored extensively to maximize data locality for the traditional single-core processors. The classical approaches are either based on array restructuring [12], or based on loop transformation [13]. Recently, they have been extended to support many-core architectures [14], [15]. In this work, rather than re-inventing the wheel, we analyze the effects of two existing vectorization approaches and their portability for OpenCL kernels in the context of different architectures and different MAPs. Based on this analysis, we aim to further refine (rather than generalize) the optimizations and make them easily tunable with the target platforms and MAPs.

# 7 Conclusions and Future Work

In this paper, we provide two strategies to vectorize naive OpenCL kernels: *inter-vectorization* and *intra-vectorization*. We evaluate the bandwidth changes of several MAPs in the context of vectorization, and we show an improvement in the overall kernel execution time on both SIMD and non-SIMD architectures. We conclude that vectorization leads to better platform utilization for both memory access and computational power. Finally, an extended framework is presented taking data with proper data layouts as input to maximize performance. Our immediate future work will focus on implementing the CLVectorizer framework.

## References

[1] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. 4, 6

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, September 2006. 4

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, October 1987. 4, 15

[4] AMD Inc. AMD Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/gpu/amdappsdk/pages/default.aspx, Februay 2011. 4, 7

[5] Intel Inc. Practical Optimization with AVX, 2011. 4

[6] Intel Inc. Intel OpenCL SDK. http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/, 2011. 4

[7] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, Februay 2011. 5

[8] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28:363–400, August 2000. 5, 15

[9] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense Two-Frame stereo correspondence algorithms. *Int. J. Comput. Vision*, 47(1-3):7–42, April 2002. 11

[10] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*. Stanford University, August 1997. 16

[11] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00 )*, volume 35, pages 145–156, New York, NY, USA, May 2000. ACM. 16

[12] Shun-tak Leung and John Zahorjan. Optimizing data locality by array restructuring. Technical report, University of Washington, September 1995. 16

[13] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, May 1991. 16

[14] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in Data-Parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, January 2011. 16

[15] Shuai Che, Jeremy Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC'11)*, November 2011. 16