Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels

Jianbin Fang, Henk Sips
Delft University of Technology
Email: {j.fang, h.j.sips}@tudelft.nl

Pekka Jääskeläinen
Tampere University of Technology
Email: pekka.jaaskelainen@tut.fi

Ana Lucia Varbanescu
University of Amsterdam
Email: a.l.varbanescu@uva.nl

***Abstract*—Due to the diversity of processor architectures and application memory access patterns, the performance impact of using *local memory* in OpenCL kernels has become unpredictable. For example, enabling the use of local memory for an OpenCL kernel can be beneficial for the execution on a GPU, but can lead to performance losses when running on a CPU. To address this unpredictability, we propose an empirical approach: by disabling the use of local memory in OpenCL kernels, we enable users to compare the kernel versions *with* and *without* local memory, and further choose the best performing version for a given platform.**

To this end, we have designed *Grover*, a method to automatically remove local memory usage from OpenCL kernels. In particular, we create a *correspondence* between the global and local memory spaces, which is used to replace local memory accesses by global memory accesses. We have implemented this scheme in the LLVM framework as a compiling pass, which automatically transforms an OpenCL kernel with local memory to a version without it. We have validated Grover with 11 applications, and found that it can successfully disable local memory usage for all of them. We have compared the kernels with and without local memory on three different processors, and found performance improvements for more than a third of the test cases after Grover disabled local memory usage. We conclude that such a compiler pass can be beneficial for performance, and, because it is fully automated, it can be used as an auto-tuning step for OpenCL kernels.

***Index Terms*—OpenCL, Local Memory, Reverse Engineering.**

I. Introduction

Multi-core processors and/or many-core accelerators are nowadays present in virtually any computing machine, from supercomputers to desktops and laptops. As the number of processing cores on a single chip is still growing, concurrent memory requests lead to increased contention for memory. Therefore, architects build complex memory hierarchies and constantly increase memory bandwidth to limit the performance effects of such contention. Hardware caches remain a popular solution, but not the only one. For example, modern GPUs use localized (i.e., per group of cores) programmer-managed *scratch-pad memories (SPMs)* together with hardware caches.

Built by the Khronos Group[1] to address the need for portability when programming modern multi/many-cores architectures, *OpenCL* (Open Computing Language) [1] relies on a common platform model that resembles, as much as possible, the real platforms. Therefore, besides processing elements and a global memory, this platform model includes such localized programmable memories under the name of *local memory*.

To fully support OpenCL, platform vendors must map this abstract platform model on their real hardware. Therefore, different vendors have mapped local memory on top of their hardware and software stack in different manners. For example, NVIDIA GPUs map local memory onto the on-chip SPM (called *shared memory* in CUDA). Multicore CPUs, which are cache-only processors[2], map local memory onto the off-chip memory [2].

For application implementation, OpenCL programmers typically make explicit use of local memory in the attempt to gain performance, especially when programming GPUs. However, the architectural diversity of the underlying platforms makes the performance impacts of using local memory unpredictable [3], [4]: enabling local memory usage for the same application can lead to performance improvements for (some) GPUs and performance losses for other GPUs and/or (some) CPUs (Section II).

In this work, we propose an empirical solution to address this unpredictability: by disabling the use of local memory for OpenCL kernels, programmers can make a direct performance comparison between the two versions of a kernel (with and without local memory), and choose the best performing version for a given platform.

However, disabling the local memory usage requires in-depth code analysis for systematic address translation between the global and local memory spaces. Such a "reverse engineering" process is always error-prone and time-consuming, and it is especially difficult when kernels are very complex and/or designed by a third party. Instead, an automated tool is desirable to facilitate these changes. Moreover, by embedding such a tool in a compiler, the choice between kernels with and without local memory can easily become a performance auto-tuning step, and can enable code specialization for performance portability [5].

To this end, we propose Grover, a method to automatically disable local memory in OpenCL kernels, and implement it as a compiler pass. The key challenge for Grover is to create a correspondence between the accesses from the

[1]http://www.khronos.org

[2]Cache-only processors have on-chip hardware caches, but no programmable SPMs.

local and global memory spaces. We show how we built this correspondence (Section III) and how it can be used in the LLVM compiling framework as an optimization pass (Section IV). We evaluate our approach on 11 applications (Section V) and show that Grover is able to transform all of them. Moreover, we obtain interesting performance numbers on three different platforms: by disabling local memory, 36% of the test cases show performance improvement, while 27% of them suffer a performance loss (Section VI). Overall, we conclude that Grover is a first auto-tuning prototype that can transparently alleviate performance losses due to ineffective, platform-unfriendly usage of local memory.

Most previous work [6]–[13] focuses on enabling SPMs or local memory. However, we believe that in the context of performance portability over diverse processors and their different local memory implementations, "reversing" local memory usage becomes equally important to enabling it. To the best of our knowledge, our work is the first study on disabling the use of local memory in OpenCL kernels. Although we focus on OpenCL in this paper, our approach is similarly applicable to CUDA and shared memory, which can show performance losses for different generations of GPUs [3]. To summarize, the contributions of our work are as follows:

- We propose a formal approach to determine the correspondence between global and local memory spaces for a given OpenCL kernel.
- We present and demonstrate an empirical approach to detect the usage of local memory in an OpenCL kernel.
- We describe Grover, a method for disabling the use of local memory in OpenCL kernels, and its implementation as a compiler pass based on LLVM.
- We empirically prove the usability of Grover as a performance auto-tuning tool on a set of 33 test-cases (11 applications on 3 different platforms).

II. Background and Motivation

In this section we introduce OpenCL and its memory model. We also show that disabling local memory usage in OpenCL kernels can lead to unexpected performance gain, a conclusion that further motivates our work.

A. OpenCL and Local Memory

OpenCL is a relatively new standard for parallel programming of heterogeneous systems [1]. Addressing functional portability, OpenCL uses a generic platform model comprising a host and one or several devices, which are seen as the computation engines. Devices have multiple *processing elements* (PEs), further grouped into several *compute units* (CUs), a *global memory* and *local memories*.

OpenCL programs have two parts: *kernels* that execute on one or more devices, and a *host program* that executes on the host (typically a traditional CPU). The host program defines the *contexts* for the kernels and manages their execution, while the computational task is coded into kernel functions. When a kernel is submitted to a device for execution, an index space of *work-items* (instances of the kernel) is defined. Work-items, grouped in *work-groups*, are executed on the processing elements of the device in a lock-step fashion. Each work-item has its own *private memory* space, and can share data via the *local memory* with the other work-items in the same work-group. All work-items can read/write the *global memory*.

B. Disabling Local Memory Usage

Applications written in OpenCL are functionally portable: the same application runs correctly on different hardware platforms. However, the optimization strategies differ over platforms. For example, GPU programming guides recommend enabling the use of local memory as a performance booster [14], while CPU programming guides argue for avoiding it [15]. Thus, migrating GPU-optimized code to CPUs might require some degree of code specialization [5], especially when the performance penalties are significant.

Disabling local memory usage is such a code specialization, which requires programmers to analyze kernel code, locate the candidate data structures that are placed and accessed in local memory, perform address translation between the local and global memory spaces, and remove redundant instructions. Doing the reversing work manually can be error-prone and time-consuming, which is particularly true in a complicated program context.

C. Performance Impact

To illustrate how significant the performance impact of disabling local memory is on different platforms, we use two benchmarks from the NVIDIA SDK: MT– `Matrix Transpose` and MM– `Matrix Multiplication`. We compare their performance on 6 platforms (including GPUs, CPUs, and an Intel Xeon Phi described in Section V-C).

In the original MT code (Figure 1(a), line 6), local memory is used to stage data (i.e., cache it `in`). To disable local memory usage, we identify the candidate data structure (Figure 1(a), line 5), manually substitute the local memory access with its corresponding global memory access (Figure 1(b), line 10) and remove the redundant instructions (Figure 1(b), lines 5-8), including data structure declarations and barriers. For MM, which calculates $C(i,j) = A(i,k) \times B(k,j)$, we manually remove the local memory usage for matrix A, while keeping it enabled for matrix B.

The performance impacts of these transformations, for both MT and MM, are presented in Figure 2. For MT, removing the local memory usage leads to performance losses on GPUs (Fermi, Kepler, and Tahiti), but improves performance for the cache-only processors (SNB, Nehalem, and MIC). The performance increase is up to 1.3× on SNB and 1.6× on Nehalem. For MM, by disabling local memory, we achieve a better performance on Tahiti, SNB (1.6×), and MIC, but we lose performance on the other three processors.

These results show that removing local memory usage can lead to (significant) performance improvement, but the cases when improvements appear are not as predictable as expected (i.e., the rule "local memory for GPUs, no local memory

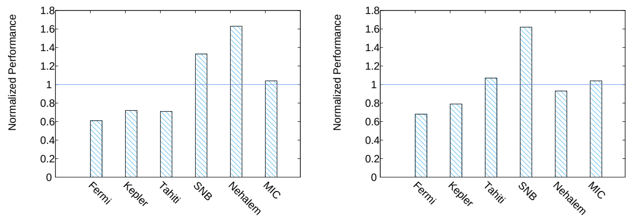
```
1  __kernel void
2  MatTrans(const __global float * in, \
3    __global float * out, int W, int H){
4
5    __local float lm[S][S];
6    lm[ly][lx]=in[(wx*S+ly)*W+(wy*S+lx)];
7
8    barrier(CLK_LOCAL_MEM_FENCE);
9
10   float val = lm[lx][ly];
11   out[gy * H + gx] = val;
12 }
```

(a) Original code.

```
1  __kernel void
2  MatTrans(const __global float * in, \
3    __global float * out, int W, int H){
4
5    //__local float lm[S][S];
6    //lm[ly][lx]=in[(wx*S+ly)*W+(wy*S+lx)];
7
8    //barrier(CLK_LOCAL_MEM_FENCE);
9
10   float val = in[(wx*S+lx)*W+(wy*S+ly)];
11   out[gy * H + gx] = val;
12 }
```

(b) Remove local memory.

Fig. 1: Removing local memory usage on Matrix Transpose. (lx, ly) is the local work-item index, (wx, wy) is the work-group index, (gx, gy) is the global work-item index, (W, H) is the global data size, (S, S) is the local data size.



(a) Matrix Transpose (b) Matrix Multiplication

Fig. 2: The performance impacts of removing local memory on two applications (the normalized performance is ratio of the performance without local memory to that with local memory).

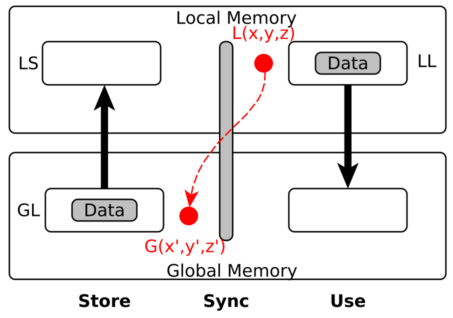


Fig. 3: A common pattern for using local memory. The figure is simplified to only present the two stages of interest for Grover and the main operations that affect the translation.

for CPUs" does not always hold). Therefore, we argue that, performance-wise, disabling local memory usage becomes a significant optimization for OpenCL kernels, especially in the context of inter-platform portability. In the remainder of this paper, we show how this optimization can be applied automatically.

III. Grover: Systematically Disabling Local Memory Usage

In this section, we present the challenges of disabling local memory, we introduce our method, and give a brief proof for it. We finally demonstrate our method on a practical example.

A. Overview

Understanding how local memory is used - i.e., the typical patterns that appear in OpenCL kernels, allows us to systematically reverse the process, step-by-step. In this paper, we focus on the most common use-case, when local memory is used as a software-controlled cache. Specifically, this pattern has two stages of interest (seen in Figure 3): *data storage* and *data usage*. In the *storage* stage, data is loaded from global memory (global load operation, *GL*) and stored into a data structure in local memory (local store operation, *LS*), as shown in Figure 1(a), line 6. The *usage* stage refers to computation performed on/with the data stored in local memory (local load operation, *LL*), as shown Figure 1(a), line 10. A synchronization (by local barrier) is required between these stages (Figure 1(a), line 8).

When disabling local memory, we need to determine which data element in the global space (x', y', z') corresponds to a given data element in the local space (x, y, z) (Figure 3). Hence, the challenge is to determine a correspondence function between the global space and the local space.

Terminology: For readability purposes, we define a brief list of important terms commonly used in this paper.

Thread Index: OpenCL defines an N-dimensional thread (or work-item) index space ($N \le 3$). Each work-item in this space has a unique local thread index (lx, ly, lz), a global thread index (gx, gy, gz), and a work-group index (wx, wy, wz). Note that a global thread index can be calculated from the local thread index and the work-group index.

Data Index: In OpenCL, data buffers are collection of data elements, each identified by a *data index*. Data elements stored in the *local memory space* are qualified by *__local*, and will be further called *local data elements*, indexed by a local index (x, y, z). Similarly, data elements stored in *global memory space* are qualified by *__global*, and will be called global data elements (indexed by a global data index, (x', y', z')). Our goal is to find a systematic correspondence between the global (memory) space and the local (memory) space, such that we can determine the global data index corresponding to a given local memory access.

Typically, each thread works on data elements in a region determined by its thread index. Hence, a *data index* is a function of a *thread index*: a local data index is a function of a local thread index (lx, ly, lz), while the global data index is a function of both the local thread index (lx, ly, lz) and the work-group index (wx, wy, wz).

B. The Method behind Grover

Before disabling local memory, we need to select all the candidate data structures by analyzing the kernel code. We assume for now these candidates are known (the details on determining them are mentioned in Section IV-A), and we focus on determining the global data index with the following steps.

S1. Analyze the local memory accesses (*LS* and *LL*) and determine the local data index for both *store* and *load* operations. Here we represent the *LS* data index as (x, y, z) and the *LL* data index as (x_{LL}, y_{LL}, z_{LL}). Given that local data index is a function of local thread index, we obtain Equation 1.

$$\begin{cases} x = f(lx, ly, lz) \\ y = g(lx, ly, lz) \\ z = h(lx, ly, lz) \end{cases} \tag{1}$$

where we assume f, g, and h are linear functions of lx, ly, and lz. Thus, we further substitute these functions in Equation 1 to obtain Equation 2.

$$\begin{cases} x = a_0 \cdot lx + b_0 \cdot ly + c_0 \cdot lz + d_0 \\ y = a_1 \cdot lx + b_1 \cdot ly + c_1 \cdot lz + d_1 \\ z = a_2 \cdot lx + b_2 \cdot ly + c_2 \cdot lz + d_2 \end{cases} \tag{2}$$

where a_i, b_i, c_i, and d_i ($i \in \{0, 1, 2\}$) are constants for a local memory usage. Similarly, we can represent the *LL* data index x_{LL}, y_{LL}, and z_{LL} as a function of local thread index. However, we consider them to be constants here, and seek to determine the global data index for the local data index (x_{LL}, y_{LL}, z_{LL}).

S2. Create a linear system and solve it for (lx, ly, lz). From the previous step, we can establish a system of linear equations in Equation 3, where lx, ly, and lz are the *unknowns* and a_i, b_i, c_i, and d_i ($i \in \{0, 1, 2\}$) are the coefficients, and x_{LL}, y_{LL}, z_{LL} are the constant terms of the system. By solving the system, we obtain the solution $(\mathsf{lx}, \mathsf{ly}, \mathsf{lz})$.

$$\begin{cases} a_0 \cdot lx + b_0 \cdot ly + c_0 \cdot lz + d_0 = x_{LL} \\ a_1 \cdot lx + b_1 \cdot ly + c_1 \cdot lz + d_1 = y_{LL} \\ a_2 \cdot lx + b_2 \cdot ly + c_2 \cdot lz + d_2 = z_{LL} \end{cases} \tag{3}$$

We note that the global data index is reversible if the system has a single unique solution. Consequently, when the system does not have a unique solution, Grover will not be able to cancel the use of the local memory for that particular case.

S3. Analyze the *GL* operation to determine G, a function of the the work-group index *and* the local thread index: $(x', y', z') = G((wx, wy, wz), (lx, ly, lz))$.

S4. Substitute the solution of the system in G to find the new global index. $G((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{lz}))$ is then the global data index corresponding to the local data index (x_{LL}, y_{LL}, z_{LL}).

Proof. By analyzing the process of loading data elements from the global space to the local space, we establish a correspondence G (G is determined for a memory access).

$$(x, y, z) \to \mathsf{G}(x, y, z). \tag{4}$$

With Equation 1 and the global data index expression (in *S3*), we derive Equation 5 from Equation 4.

$$\begin{aligned} &(f(lx, ly, lz), g(lx, ly, lz), h(lx, ly, lz)) \\ &\qquad \to \mathsf{G}((wx, wy, wz), (lx, ly, lz)). \end{aligned} \tag{5}$$

Given a local data index (x_{LL}, y_{LL}, z_{LL}), we obtain

$$(x_{LL}, y_{LL}, z_{LL}) \to \mathsf{G}((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{lz})),$$

where the work-group part (wx, wy, wz) stays the same for the threads within a work-group and $(\mathsf{lx}, \mathsf{ly}, \mathsf{lz})$ is the solution to Equation 3. Therefore, $\mathsf{G}((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{lz}))$ is the global data index of the local data index (x_{LL}, y_{LL}, z_{LL}). $\square$

C. An Example: Matrix Transpose

To illustrate how Grover's method works, we discuss the Matrix Transpose example (see Figure 1(a)). Furthermore, to bridge to the implementation phase (Section IV), we introduce *index expression trees* (Figure 4 and Figure 5) as a notation for data indexes. In an expression tree, the leaves are operands, such as constants or variable names, and the internal nodes contain operators, such as additions $(+)$ and multiplications $(*)$. By applying the operator to the operands, we evaluate an index expression tree and obtain the data index.

S1. By analyzing the code in Figure 1(a), we abstract the *LS* data index as (lx, ly) and the *LL* data index as (ly, lx). Accordingly, their expression trees are shown in Figure 4. Note that S is the width of the allocated local data space.

S2. We know that,

$$LS : \begin{cases} x = lx \\ y = ly \end{cases}, \qquad LL : \begin{cases} x_{LL} = \underline{ly} \\ y_{LL} = \underline{lx} \end{cases},$$

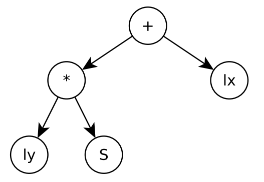
where we underline the *LL* data index (constant terms of Equation 3) to differentiate it from the *LS* data index (unknowns of Equation 3). We then create a system of linear equations as,

$$\begin{cases} lx = \underline{ly} \\ ly = \underline{lx} \end{cases}$$

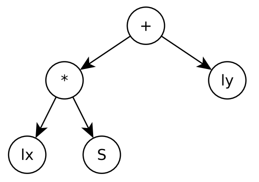
It is straightforward to get the solution of this system, i.e., $(\mathsf{lx}, \mathsf{ly}) = (\underline{ly}, \underline{lx})$.

S3. By analyzing the code in Figure 1(a), we obtain that the *GL* data index is $((wy, wx), (lx, ly))$ and its index expression tree is shown in Figure 5(a). We note that, due to a more complicated index composition (work-group index and local thread index), the index tree has more levels than the local index tree (see Figure 4).

S4. We update the global data index with the solution $(\underline{ly}, \underline{lx})$ and obtain $((wy, wx), (\underline{ly}, \underline{lx}))$, which is the new global load index shown in Figure 5(b).

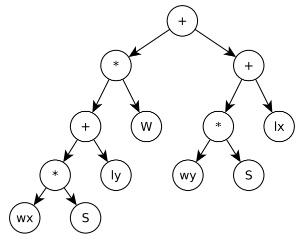


(a) Local store.

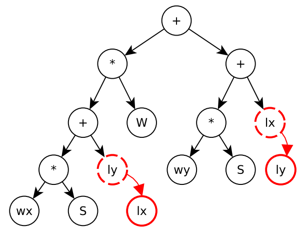


(b) Local load.

Fig. 4: Local access data index in expression tree.



(a) Global load.



(b) New global load.

Fig. 5: Global load data index in expression tree.

IV. Grover Implementation

Ideally, *Grover* transforms any OpenCL kernel that uses local memory into a version without local memory usage. To achieve this goal, we need to create the new global load instruction *nGL*, and its index-related instructions. Therefore, we implement the reversing algorithm (in Section III-B) in the following six steps:

1) Selecting the reversing candidates (Section IV-A).
2) Building the index expression trees (Section IV-B).
3) Determining the data index (Section IV-C).
4) Creating and solving the linear system (Section IV-D).
5) Duplicating the new load instruction (Section IV-E).
6) Updating the new expression tree (Section IV-F).

A. Selecting Candidates

Before removing local memory usage, we need to select the candidate data structures and detect the three operations: *GL*, *LS*, and *LL*. To do so, we first investigate all the *GL* operations in the kernel and check whether their paired store operations are *LS* operations. Next, we locate all *LL* operations that use the same data structures as the identified *LS* operations.

We note that there are applications - such as `image convolution`, for example - where multiple passes are required to load data from global memory to local memory. This means that the detection phase will identify more (*GL*, *LS*) pairs. However, using any of the pairs leads to the same correspondence between the global and the local space. Hence, we can choose any one of these pairs.

B. Building the Index Expression Trees

To enable the transformations mentioned in Section III-B, we use the *index expression tree* to represent a data index, as introduced in Section III-C. Figure 6 shows the tree node data structure, which has four fields: (1) the *value* field, (2) the *state* field, (3) the pointers to its children nodes, and (4) the pointer to its parent node. The *value* field can be an instruction, a built-in function, a constant number, or an argument. The *state* field is designed for a special requirement: to mark whether the current node needs to update the data index. To facilitate tree traversing, the structure also contains pointers to its children nodes and its parent node.

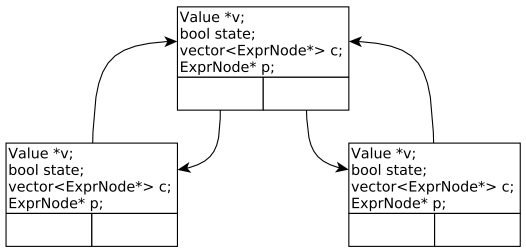


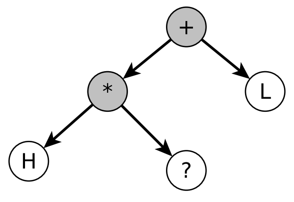
Fig. 6: Tree node structure (*ExprNode*).

The index expression tree for a memory access is built recursively. The internal nodes of an index tree can have one child like a *type cast instruction*, or two children like an *addition instruction*. Thus, an index expression tree can have one (or two) sub-tree(s). The recursive algorithm stops when the *value* is one of the follows: (1) a *call instruction*, (2) a *constant number*, (3) a *function argument*, or (4) a *phi node*. In this way, we can build the index expression trees *GLTree*, *LSTree*, and *LLTree* for *GL*, *LS*, and *LL*, respectively.

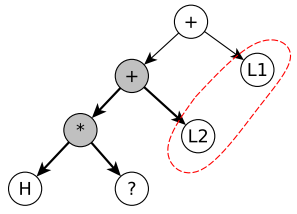
C. Determining the Data Index

To create the linear equations (in *S2* of Section III-B), we need to specify the data index of *LS* and *LL* from their index expression trees *LSTree* and *LLTree*. In this paper, we use a pattern (Figure 7(a), a 2D example) to identify the data index: when traversing an index tree top-down, we first find the '$+$' node (a node with an addition instruction), which splits the high dimension (*H*) and the low dimension (*L*). The high dimension is further identified by the '$*$' node (a node with a multiply instruction). Note that it can also be a *shift-left* operation instead of a '$*$' operation. When it comes to a 3D example, we use a similar way to determine the *LS* data index (x, y, z) and the *LL* data index (x_{LL}, y_{LL}, z_{LL}).

In real-world cases, the pattern can be more complicated as shown in Figure 7(b), where `L1` is a loop-dependent term while the others are independent of the loop. Therefore, the `L1` term lies at the second-level of the tree, to avoid computing the loop-independent terms repeatedly. However, the '$+ \rightarrow *$' pattern (in Figure 7(a)) remains the same. We consider the '$+ \rightarrow + \rightarrow *$' pattern as a derived one and use a special case



(a) Data index pattern.



(b) A derived pattern.

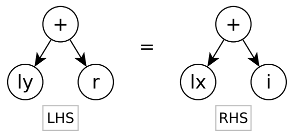
Fig. 7: Data index patterns.

to handle it. In this way, we get the low dimension of the data index as $L1 + L2$ (see Figure 7(b)).

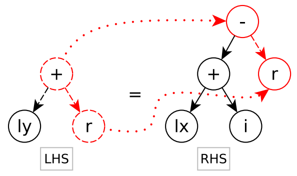
D. Creating and Solving the Linear System

Based on the data index of *LS* and *LL* obtained in the previous step, we can create a system of linear equations, each of which has a left-hand-side (LHS) term and a right-hand-side (RHS) term. To obtain (lx, ly, lz) from the system, we need to simplify the LHS term. At the same time, complementary operations are required on the RHS term.

Figure 8 shows an example of how we simplify the LHS terms. The LHS term is $(ly + r)$ and the RHS term is $(lx + i)$. By subtracting r from both sides, we can obtain ly on the LHS and get the RHS expression tree $((lx+i)-r)$ (Figure 8(b)). Similarly, other complementary operations are equally required for a more complicated linear system.



(a) Initial form.



(b) Final form.

Fig. 8: Simplifying the left-hand-size terms.

E. Duplicating the New Load Instructions

Creating a new global load instruction (*nGL*) includes creating the load instruction itself and the instructions for the calculation of its index-related instructions. We need to prepare these two parts when replacing the *LL* instruction. Since the original global load operations (*GL*) might be used somewhere else in the code (other than the one used by the local store), it is not safe to re-use and update *GLTree*. Hence, we need to duplicate *GL* (as *nGL*) and its index-related instructions.

To do so, two sub-steps are required: (1) update the state of each tree node, and (2) create the index-related instructions. First, we mark the nodes that need instruction duplication in *GLTree*. To this end, we locate the lx, ly, and lz which are the nodes to be replaced. From there, all the nodes preceding

Algorithm 1: Duplicating instructions algorithm.

```
input  : A pointer to the current node $node$
input  : A position to insert the instruction $pos$
output : A pointer to the newly created value $v$
duplicateInst (node, pos)
begin
    $v \leftarrow$ getValue (node);
    if $isCallInst(v)$ or $isConst(v)$ or $isArgs(v)$ or
    $isPHI(v)$ then
        if $node.state$ then
            $oldInst \leftarrow$ getInst (v);
            $newInst \leftarrow$ cloneInst (oldInst);
            insertInst (newInst, pos);
            $return\ newInst$;
        $return\ v$;
    else
        $nChildren \leftarrow$ getNumOfChildren (node);
        if $nChildren == 1$ then
            $child \leftarrow$ getChild (node, 0);
            $vL \leftarrow$ duplicateInst (child, pos);
            if $node.state$ then
                $oldInst \leftarrow$ getInst (v);
                $newInst \leftarrow$ cloneInst (oldInst);
                insertInst (newInst, pos);
                setOperand (newInst, vL);
                $return\ newInst$;
            $return\ v$;
        if $nChildren == 2$ then
            $lChild \leftarrow$ getChild (node, 0);
            $rChild \leftarrow$ getChild (node, 1);
            $vL \leftarrow$ duplicateInst (lChild, pos);
            $vR \leftarrow$ duplicateInst (rChild, pos);
            if $node.state$ then
                $oldInst \leftarrow$ getInst (v);
                $newInst \leftarrow$ cloneInst (oldInst);
                insertInst (newInst, pos);
                setOperand (newInst, vL);
                setOperand (newInst, vR);
                $return\ newInst$;
            $return\ v$;
```

these nodes are marked as *to be updated*. We backtrack the tree until we reach the root node. We reuse the sub-expressions that are shared by the *GL* instruction and the *nGL* instruction when it is not required to update the node.

With the updated expression tree, we duplicate the instructions and insert them before the *LL* instruction. When inserting instructions into the kernel, special care on the expression construction order is needed. Here we use the post-order DFS approach to traverse the tree and create the required instructions (Algorithm 1). Note that we need to set the *use-and-value* relationship between instructions. In this way, we can create the *nGL* and its index calculation instructions, while keeping the *GL* instruction.

F. Updating the New Expression Tree

Once we got the *nGL* instruction, a new index expression tree *nGLTree* is built with the method mentioned in Section IV-B. Thereafter, starting with the root node of *nGLTree*, we locate the lx, ly, and lz nodes and substitute them with the solution $(\mathsf{lx}, \mathsf{ly}, \mathsf{lz})$ obtained in Section IV-D. Note that type casting instructions might be required when performing the substitution. Finally, all the uses of the *LL* instruction are replaced with the *nGL* instruction.

V. Experimental Setup

This section discusses how we incorporate our compiler pass with vendors' run-time, the benchmarks, and the devices used in the experiments.

A. Incorporating Grover

To allow Grover to be fully automated, we have implemented it in a LLVM/Clang (v3.2) compiling framework [16]. The pipeline is shown in Figure 9. Taking OpenCL C kernels, the LLVM front-end (Clang) transforms them into the SPIR [17] format (LLVM IR format). Thereafter, *Grover* analyses the SPIR code and removes local memory usage (when detecting any). Our framework then exports the SPIR kernels into vendor-specific run-time such as Intel SDK for OpenCL Applications 2013.

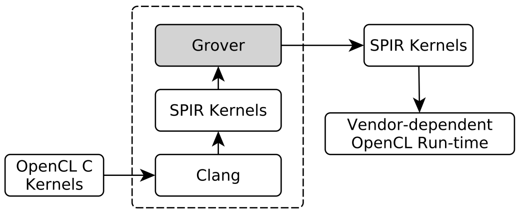


Fig. 9: Incorporating our grover.

B. Selected Benchmarks

For evaluating Grover, we selected 11 applications from the AMD SDK, the NVIDIA SDK, the Rodinia benchmarking suite and the Parboil suite. The applications and the datasets we used in the experiments are listed in Table I. All these applications are making use of local memory in their original versions. We use Grover to disable local memory usage and compare the performance of the kernels (the average of the kernel execution time over 20 runs) before and after Grover. The `oclMatriMul` application is a special case: the SDK uses local memory on two data structures (`Matrix A` and `Matrix B`). We remove them one by one, obtaining three versions of the code: `NVD-MM-A` (removing local memory for `Matrix A`), `NVD-MM-B` (removing local memory for `Matrix B`), and `NVD-MM-AB` (removing both).

Finally, we note that although the choice of the work-group size has a significant impact on the benchmark performance [18], selecting the optimal work-group size is beyond the scope of this paper (i.e., we focus on the local memory disabling, not on optimizing the overall performance of either of the kernel versions). Therefore, all the experiments use the default work-group size settings, as specified in the original benchmarks.

TABLE I: OpenCL benchmarks.

Benchmark	**Source**	**ID**	**DataSet**
StringSearch	AMD SDK	AMD-SS	StringSearch_Input.txt
MatrixTranspose	AMD SDK	AMD-MT	10240x10240
RecursiveGaussian	AMD SDK	AMD-RG	RecursiveGaussian_Input.bmp
MatrixMultiplication	AMD SDK	AMD-MM	2048x2048x2048
oclTranspose	NVIDIA SDK	NVD-MT	10240x10240
oclMatrixMul	NVIDIA SDK	NVD-MM-A	A(800 x 1600), B(800 x 800)
oclMatrixMul	NVIDIA SDK	NVD-MM-B	A(800 x 1600), B(800 x 800)
oclMatrixMul	NVIDIA SDK	NVD-MM-AB	A(800 x 1600), B(800 x 800)
oclNbody	NVIDIA SDK	NVD-NBody	163840
stencil	Parboil	PAB-ST	small
streamcluster	Rodinia	ROD-SC	(10 20 256 65536 65536 1000)

C. Platforms and Devices

At the moment of writing [3], only Intel has released an OpenCL implementation supporting SPIR [17]. NVIDIA and AMD have no SPIR support yet. Thus, we have run and compared the benchmarks on three devices from Intel (Nehalem, SNB, and MIC), whose configurations are shown in Table II. While this selection of devices might lead to a certain bias in the performance gain/loss ratio after using Grover (as CPUs are likely to benefit more from disabling local memory [15] than GPUs), it does not challenge the correctness of this proof-of-concept: the kernels Grover builds will be portable and thus execute correctly on any devices that support SPIR.

TABLE II: The platforms used for our experiments. The GPU devices do not support SPIR, and were only used when manual kernel transformations were applied (Section II)

.

Name	**Fermi**	**Kepler**	**Tahiti**
Host	Intel Xeon E5620	Intel Xeon E5620	Intel Xeon E5620
Host OS	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	NVIDIA Tesla C2050	NVIDIA Tesla K20m	AMD HD7970
GCC	v4.4.6	v4.4.6	v4.4.6
OpenCL	CUDA v5.5	CUDA v5.5	AMD APP v2.8
SPIR	No support	No support	No support
Name	**SNB**	**Nehalem**	**MIC**
Host	Intel Xeon E5-2620	Intel Xeon X5650	Intel Xeon E5-2620
Host OS	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	Intel Xeon E5-2620	Intel Xeon X5650	Intel Xeon Phi 5110P
GCC	v4.4.6	v4.4.6	v4.4.6
OpenCL	Intel OCL SDK v3.2.1	Intel OCL SDK v3.2.1	Intel OCL SDK v3.2.1
SPIR	ver 1.2	ver 1.2	ver 1.2

VI. Performance Evaluation and Discussion

In this section, we first use Grover to test whether we can disable local memory usage for each benchmark. We also evaluate the performance impact of this transformation and discuss the method limitations.

A. Calculating the New Data Index

Table III shows the data index of *nGL* for each benchmark. We first abstract the index of *GL*, *LS*, and *LL*. With the approach proposed in Section III-B, we calculate the data index of *nGL*. For `AMD-SS`, `NVD-NBody`, and `ROD-SC`,

[3]February, 2014.

the work-group index is zero. This is because all the work-items share the same data block. For example, the pattern string in `AMD-SS` is shared by all the work-items. Note that *wx, wy, wz, lx, ly, lz* represent the work-group and work-item indexes; all the other symbols are application specific. After the transformation, each benchmark still runs correctly, proving the correctness of the changes.

TABLE III: Determining the data index of *nGL*.

ID	GL	LS	LL	nGL
AMD-SS	((0, 0, 0), (lx, 0, 0))	(lx, 0, 0)	(i, 0, 0)	((0, 0, 0), (i, 0, 0))
AMD-MT	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(lx, ly, 0)	((wy, wx, 0), (lx, ly, 0))
NVD-MT	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(ly, lx, 0)	((wx, wy, 0), (ly, lx, 0))
AMD-RG	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(lx, ly, 0)	((wx, wy, 0), (lx, ly, 0))
AMD-MM	((wx, wy, 0), (lx+i*S, ly, 0))	(lx, ly, 0)	(j, ly, 0)	((wx, wy, 0), (j+i*S, ly, 0))
NVD-MM-A	(lx+a, ly, 0)	(lx, ly, 0)	(k, ly, 0)	(k+a, ly, 0)
NVD-MM-B	(lx+b, ly, 0)	(lx, ly, 0)	(lx, k, 0)	(lx+b, k, 0)
NVD-MM-AB	–	–	–	–
NVD-NBody	((0, 0, 0), (i+lx, 0, 0))	(lx, 0)	(_i, 0)	((0, 0, 0), (i+_i, 0, 0))
PAB-ST	((wx, wy, 0), (lx, ly, k))	(lx, ly, 0)	(lx, ly, 0)	((wx, wy, 0), (lx, ly, k))
ROD-SC	((0, 0, 0), (x, lx, 0))	(lx, 0, 0)	(i, 0, 0)	((0, 0, 0), (x, i, 0))

B. Results Summary

The performance results on SNB, Nehalem, and MIC are shown in Figure 10. The base-line performance is obtained when using local memory. The normalized performance (np) is the ratio of the performance without local memory to that with local memory. When this ratio is close to 1 (within 5%), the two versions of the kernel have *similar* performance. For ratios below 1, Grover's pass leads to *performance losses*, while for ratios larger than 1 we speak about *performance improvement*.

We show the overall performance gain/loss distribution for a similarity threshold of 5% in Table IV. In total, for our 33 test-cases, 36% benefit from disabling local memory, while 27% show performance loss.

TABLE IV: Performance gain/loss distribution

	SNB	Nehalem	MIC	Total (%)
Gain	6	4	2	12 (36%)
Loss	2	4	3	9 (27%)
Similar	3	3	6	12 (36%)

C. Performance Analysis

We further analyze the performance results on SNB, Nehalem, and MIC, as shown in Figure 10. We observe that disabling local memory leads to varied performance results. On SNB, we observe speedups of 1.67×, 1.12×, 1.18×, 1.07×, 1.16× for `NVD-MT`, `AMD-RG`, `NVD-MM-A`, `NVD-MM-AB`, `PAB-ST`, respectively. The kernel performance drops by 44% for `AMD-MM`, 19% for `NVD-MM-B` and 5% for `NVD-NBody`. For `AMD-SS`, `AMD-MT`, and `AMD-RG`, the performance is only marginally affected.

We noticed a significant performance increase for `NVD-MT` on SNB. To ensure all global reads and writes are coalesced, `NVD-MT` uses local memory to stage data on GPUs. On CPUs, however, the coalescing rules are not necessary. Further, the work-items within a work-group are usually mapped onto a hardware thread [2], which is a kind of *tiling* and implicitly enables data locality. The reason for the performance increase is the same for `AMD-RG`. For `AMD-MT`, removing local memory brings little gain due to the explicit usage of vector data types (i.e., each work-item works on 4×4 matrix elements).

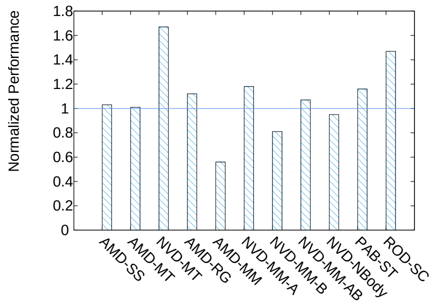
For `NVD-MM-A`, SNB shows performance improvement. When analyzing the memory access of `Matrix A`, we noticed that each work-item needs a row of data elements from it. On GPUs, this generates a large amount of data reuse among the work-items of the same work-group. When it comes to CPUs, this data reuse is implicitly exploited by the on-chip caches. That is, the data elements (in the form of cache-lines) loaded by a work-item can be reused by its neighboring work-items. In this way, using local memory itself becomes an extra overhead, compared with only using the caches. Therefore, removing local memory gives a performance increase. The explanations stay the same for `PAB-ST`.

We noticed a performance drop on `AMD-MM` and `NVD-MM-B` by removing local memory usage. We found that the application uses local memory on the column-wise accessed matrix. For such a case, using local memory changes the data layout, and ensures that data elements are reused before they are kicked out of caches. Meanwhile, `AMD-MM` uses data in a row-major manner, but it exploits vector data types, which changes the memory access pattern to be column-major. Thus, removing local memory is also detrimental to `AMD-MM`'s performance.

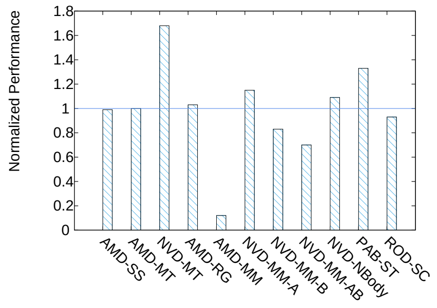
By removing local memory, we see a performance drop on `NVD-NBody` on SNB, but a slight speedup on Nehalem and MIC. This happens because in `NVD-NBody`, each work-item needs to access all the input data elements (bodies). Thus, the work-items within a work-group will access the same element simultaneously. The pattern should be identified by caches and therefore we expect that removing local memory should give a performance increase (the reasons behind the performance loss on SNB are under investigation). A similar observation can be made about `ROD-SC`: performance increases on SNB, but decreases on Nehalem and MIC. We expect removing local memory to lead to a performance boost, due to all work-items sharing a small array of 16 data elements, stored far from each other (not in a cacheline). When using local memory, these elements are gathered and stored contiguously in the local space. Thus, this case resembles `NVD-MM-B`, which gives a better cache utilization by using local memory.

In general, Nehalem and SNB show similar performance trends when we disable local memory (Figure 10(b)), with the exception of the number for `NVD-MM-AB`. MIC behaves significantly different: we observe that most applications have similar performance with and without using local memory; only minor differences can be observed for `NVD-MM-A/B/AB`. This is mainly because MIC has a different cache hierarchy: a distributed last-level cache, compared with a unified one on Nehalem and SNB. This architectural difference minimizes the performance gaps between with and without local memory.

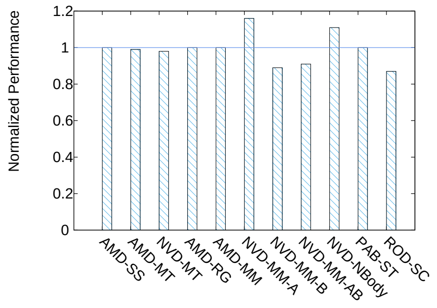
Overall, disabling local memory still leads to unpredictable performance outcomes on cache-only processors from differ-



(a) On SNB



(b) On Nehalem



(c) On MIC

Fig. 10: Normalized performance on three devices.

ent generations. Thus, the empirical exploration of Grover remains the ideal approach for choosing the best performing version of a kernel for a given platform.

D. Limitations

Grover can successfully remove local memory usage, but it has its limitations. Being built based on a common use-case (i.e., a local memory usage pattern), the tool is applicable for all kernels fitting this use-case. For other use-cases - e.g., when local memory is used as temporal storage for repeated read/write operations - the analysis must be adjusted. However, in our experience, such applications (e.g., `reductions`), typically benefit from using local memory [4] on any platform.

Furthermore, using local memory often leads to implementing tiling, and the resulting code may have a different algorithm (code skeleton) than its original (sequential) form. Grover transforms the code into a version without local memory, but not to its original form. Such a case can be seen with our tests with `NVD-NBody`, which achieves a better performance with the original form.

VII. Related Work

We list here several directions of past research that are related to this work: address translation, enabling local memory, special compiling of GPU code for CPUs, and special compiler passes for performance improvement.

Address translation between a large (global) space and a small (local) space is an old research topic in operating systems and computer architecture. For example, address translation between virtual and physical memory requires a mapping stored in a *page table*, where each entry includes flags and a frame number [19]. Similarly, to find a line in a cache, both the slot bits and tag bits are used to correlate the global address and local cache address [20]. In our work, this correspondence needs is neither fixed, nor static: it needs to be investigated and built for any kernel.

Enabling local memory has been studied extensively for the SPMs on GPUs. Most studies focus on identifying data reuse (e.g., using a polyhedral model) [8]–[12] and exploit it by enabling local memory. Alternatively, in [13], the authors present a fully automated C-to-FPGA framework, including an end-to-end solution for on-chip buffer optimization that automatically detects and implements the available date reuse in a loop nest. However, all such studies focus on data access patterns and its exploitation, while we focus on code analysis for disabling local memory.

Several API-based approaches have also been proposed to enable local memory. In [21], the authors present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memories of GPUs. In [4], the authors present a user-friendly API, ELMO, based on identifying patterns of local memory usage. We got inspired by the idea of local memory usage patterns in these papers, but our approach is fully automated for a given local memory usage pattern.

Compiling GPU kernels for CPU architectures is another old challenge. For example, MCUDA [22] is a source-to-source translator from CUDA for GPU architectures to multi-threaded C for multi-core CPU architectures. It serializes the work of a thread block within a single CPU thread and parallelizes the work of the kernel at thread block granularity. In [2], the authors present *Twine Peaks*, a software platform for heterogeneous computing that executes code originally targeted for GPUs efficiently on CPUs as well. In particular, the system maximizes the utilization of functional units in the CPUs by exploiting the data locality and data parallelism exposed by the GPGPU model through computation kernels. Neither of these systems addresses explicitly the local/shared memory issue, opting instead for larger transformations of the code.

Compiling passes for improving OpenCL's performance are being increasingly popular in search of performance portability. In [23], Yang et al. propose a source-to-source translator (based on the Cetus compiler framework) to address two major challenges: effective utilization of GPU memory hierarchies and judicious management of parallelism. In [24], Ralf Karrenberg and Sebastian Hack present a language- and platform-independent code transformation that vectorizes a function given by an arbitrary control flow graph in SSA form. They present a data-flow analysis that determines which code regions have constraints for vectorization concerning alignment and consecutiveness. In [25], Alberto Magni et al.

consider *thread-coarsening* of OpenCL kernels and evaluate its effects across a range of devices based on LLVM. In this context, our approach is unique in finding a pass that has not been yet explored: the disabling of local memory.

To summarize, ours is the first study focused on automatically *disabling* the usage of local memory in OpenCL kernels through a compiler pass. While we got inspired by previous studies on compiler passes for OpenCL code optimization and/or specialization, address translation research, and local memory enabling, our simple and functional approach is novel in goal, method, and implementation.

VIII. Conclusion

While functional portability is insured by the OpenCL platform model and the efforts of the hardware vendors to properly support it, performance portability remains a challenging task. Thus, several platform-specific optimizations have to be enabled or disabled when porting kernels from one device to another. One example of such an optimization is the use of local memory, rendered unpredictable by the diversity of hardware platforms and OpenCL mappings.

In this paper, we have shown evidence that using local memory can lead to significant performance penalties for many devices - both GPU and CPU platforms. Thus, we proposed an approach for reverse-engineering OpenCL kernels with local memory. Specifically, we designed and implemented Grover, a method for automatically removing local memory usage from OpenCL kernels in search for these performance improvements. Grover is implemented as a compiler pass, which enables programmers to auto-tune the use of local memory (i.e., on/off) to obtain the best performing version of a kernel for a given platform.

We have validated Grover on a set of 11 applications, showing that the local memory usage has been correctly canceled. For more than a third of the 33 test-cases (11 applications on 3 different platforms), we observed performance improvements. Thus, Grover can be used for exploiting potential performance improvements due to removing local memory usage in OpenCL kernels. We believe Grover proves that the performance portability of OpenCL codes can be improved by automated code specialization.

In the near future, we will further investigate Grover's impact on other types of devices (e.g., GPUs). Moreover, using Grover, we want to model the performance benefits/losses due to local memory usage on CPUs. Ultimately, we aim to incorporate *Grover* into a high-level auto-tuning framework for OpenCL kernels, where code specialization is automated for different classes of platforms.

Acknowledgment

This work was partially funded by CSC (China Scholarship Council), and the National Natural Science Foundation of China under Grant No.61103014 and No.11272352.

References

[1] Khronos OpenCL Working Group, *The OpenCL Specification V1.2*, November 2012.

[2] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of PACT'10*, PACT '10, (New York, NY, USA), pp. 205–216, ACM, 2010.

[3] J. Fang, H. Sips, and A. L. Varbanescu, "Quantifying the performance impacts of using local memory for many-core processors," in *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pp. 1–10, IEEE, 2013.

[4] J. Fang, A. L. Varbanescu, J. Shen, and H. Sips, "ELMO: A User-Friendly API to enable local memory in OpenCL kernels," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*, 2013.

[5] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance traps in opencl for cpus," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*, February 2013.

[6] M. Kandemir et al., "Compiler-directed scratch pad memory hierarchy design and management," in *Proceedings of DAC*, ACM, 2002.

[7] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embed. Comput. Syst.*, vol. 5, pp. 472–511, May 2006.

[8] A. Größlinger, "Precise management of scratchpad memories for localising array accesses in scientific codes," in *Compiler Construction* (O. Moor and M. Schwartzbach, eds.), vol. 5501 of *Lecture Notes in Computer Science*, ch. 17, pp. 236–250, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[9] M. M. Baskaran et al., "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of PPoPP*, 2008.

[10] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, Jan. 2013.

[11] N. Vasilache, M. Baskaran, B. Meister, and R. Lethin, "Memory reuse optimizations in the R-Stream compiler," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), pp. 42–53, ACM, 2013.

[12] A. Konstantinidis, P. H. J. Kelly, J. Ramanujam, and P. Sadayappan, "Parametric GPU code generation for affine loop programs," in *The 26th International Workshop on Languages and Compilers for Parallel Computing*, Sept. 2013.

[13] L. N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), pp. 29–38, ACM, 2013.

[14] NVIDIA, "CUDA Toolkit 5.5." http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, July 2013.

[15] Intel Inc., *Intel OpenCL Optimization Guide*, April 2012.

[16] P. O. Jaaskelainen, C. S. de la Lama, P. Huerta, and J. H. Takala, "OpenCL-based design methodology for application-specific processors," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 223–230, IEEE, July 2010.

[17] T. K. Group", ""spir: The standard portable intermediate representation for device programs"." http://www.khronos.org/spir, 2013.

[18] S. Seo, J. Lee, G. Jo, and J. Lee, "Automatic OpenCL work-group size selection for multicore CPUs," in *Proceedings of PACT'13*, PACT '13, (Piscataway, NJ, USA), pp. 387–398, IEEE Press, 2013.

[19] A. S. Tanenbaum, *Modern Operating Systems (3rd Edition)*. Prentice Hall, 3 ed., Dec. 2007.

[20] J. Hennessy, J. L. Hennessy, D. Goldberg, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers.

[21] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), ACM, 2011.

[22] J. A. Stratton, S. S. Stone, and W. Mei, "Languages and compilers for parallel computing," in *Languages and Compilers for Parallel Computing* (J. N. Amaral, ed.), vol. 5335 of *Lecture Notes in Computer Science*, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30, Berlin, Heidelberg: Springer-Verlag, 2008.

[23] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," *SIGPLAN Not.*, vol. 45, pp. 86–97, June 2010.

[24] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of IEEE/ACM CGO'11*, CGO '11, (Washington, DC, USA), pp. 141–150, IEEE Computer Society, 2011.

[25] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), ACM, 2013.