ELMO: A User-Friendly API to Enable 
Local Memory in OpenCL Kernels

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

Abstract—Recent parallel architectures are equipped with local memory, which simplifies hardware design at the cost of increased program complexity due to explicit management. To simplify this extra-burden that programmers have, we introduce an easy-to-use API, ELMO\(^1\), that improves productivity while preserving high performance of local memory operations. Specifically, ELMO is a generic API that covers different local memory use-cases. We also present prototype implementations for these APIs and perform multiple GPU-inspired optimizations to maximize their performance. Experimental results on the NVIDIA Quadro5000 GPU show that performance is significantly improved by using ELMO on native implementations: the achieved speedup ranges from 1.3\(\times\) to 3.7\(\times\). Furthermore, using ELMO we still achieve performance comparable (if not better) with that of hand-tuned applications, while the code is shorter, clearer, and safer.

Index Terms—Local Memory, API, OpenCL, GPUs.

I. INTRODUCTION

In the last few years, multicore/manycore processors have become increasingly popular. To exploit the full benefits of the increasing number of computational units, architects need to ensure that memory bandwidth and latency are optimized [1]. Utilizing a cache hierarchy has been the traditional way to alleviate the memory bottleneck [2], but there are still various modern parallel architectures such as NVIDIA Fermi and AMD Northern Island that use a programmer-managed scratch-pad memory, referred to as local memory\(^2\).

Because local memory is situated on-chip, it is much faster than the global memory. Thus, a proper use of local memory often leads to higher memory bandwidth and thus performance improvement. Nevertheless, using local memory is an error-prone and time-consuming process. Programmers often have to manually address, in their code, challenges like (1) geometry mismatch, (2) work-items masking and binding switches, and (3) inefficient local memory organization (see Section III). We argue that when solving these problems manually, programmers waste too much time on non-computational and non-functional coding details, which hinders productivity and bloats the code.

Multiple approaches have been proposed to improve productivity while achieving high performance for parallel architectures, which can be loosely classified into (i) new languages (e.g., OptiML [3]), (ii) auto-parallelizing compilers (e.g., OpenACC [4], Mint [5]), and (iii) libraries/APIs (e.g., Thrust [6]). In all these cases, programmers are isolated, in one way or another, from the difficult implementation details related to the platform architecture: they can focus on the functional parts of the application and leave these non-functional elements to be solved by run-times, compilers, or libraries.

In our work, taking the third approach, we focus on the design and implementation of a high-level API targeting the efficient usage of local memory on modern many-core processors. As the main difficulty of these operations is the complex and dynamic nature of the binding between the threads (work-items) and the data elements in global or local memory, we propose ELMO, a collection of easy-to-use APIs that (1) present a friendly front-end to make these bindings/mappings transparent to users (see Section IV), and (2) provide implementations and perform several optimizations to ensure the efficiency of the local memory operations (see Section V).

Summarizing, our contributions are as follows:

1) We present three challenges of using local memory and thus summarize the ELMO requirements.
2) We illustrate the design of three APIs, and further provide GPU-based back-end implementations for them.
3) We evaluate ELMO’s performance against both native kernels and hand-tuned kernels.
4) We discuss the programmability and usability of ELMO, and its limitations.

Our results show that with ELMO the kernels can run by up to 3.7\(\times\) faster over native kernels and deliver matching performance with hand-tuned kernels on NVIDIA Quadro5000 (see Section VI). Using ELMO, programmers can focus solely on the functional side of the application, which improves their productivity by enabling faster and less error-prone coding (see Section VII). Furthermore, the back-end optimizations can be adapted to novel architectures, providing better opportunities to improve the performance portability of ELMO code.

II. TARGET DEVICE ARCHITECTURE

As depicted in Figure 1, ELMO assumes a system design with local memory, based on the OpenCL device architecture [7]. To achieve high bandwidth, local memory is divided into equally-sized memory banks, which are organized in such a way that successive 32-bits words are assigned to

\(^1\)The name is inspired by the friendly muppet character in Sesame Street and the abbreviation of the API: to Enable Local Memory in OpenCL kernels.

\(^2\)NVIDIA uses the term ‘shared memory’, while AMD calls it ‘local data store’. In this paper, we will use the name ‘local memory’ (and other terms) from OpenCL.
successive banks, i.e., interleaved (the right part of Figure 1). A bank conflict occurs when two or more work-items access different words in the same bank [8], [9]. Accesses that map to the same bank are serialized and serviced in consecutive cycles, resulting in performance degradation. Thus, a key to effectively use the local memory is to control the access pattern so that simultaneous accesses are mapped to different banks. The optimization work is left to programmers and needs to be carefully considered for a larger memory bandwidth, as will be further discussed in Section III.

ELMO is targeting OpenCL-compliant platforms and kernels. Due to the cross-platform capability of OpenCL, the APIs are applicable for any OpenCL-compliant devices with local memory. However, since multi-core CPUs allocate local memory on global memory space (not on-chip memory), we cannot ensure the performance benefits of using local memory on them, and thus the back-end of ELMO is, for now, targeting GPUs. Therefore, all the experiments and results included in this paper relate to GPUs.

III. ELMO REQUIREMENTS

In this section, we give a brief description of the basic operations that programmers need to do when using local memory and explain the challenges behind them. Based on this, we define our API’s requirements.

Usually, the local memory operations consist of two types: (1) data transfers between global memory and local memory, and (2) data transfers between registers (or private memory) and local memory. In theory, any work-item of a work-group can access any data element in local memory, and there is no data coherence guarantee. Thus, it becomes difficult to manage this process especially when the number of work-items increases from ‘multi’-scale to ‘many’-scale. The following are the challenging scenarios we have identified when using these two types of basic operations.

A. Challenge I: Geometry Mismatch

When accessing data in the local memory and/or when bringing data to local memory, the simple cases of 1 : 1 or 1 : n work-items per data elements are easily solved: each work-item will access exactly 1 or n data elements. However, many applications (e.g., image convolution) also need halo data - i.e., the data elements neighboring the central part (see Figure 2). This will often lead to a geometry mismatch between the work-items used to bring the data and the data elements themselves. Thus, binding work-items to data elements in an orderly fashion becomes difficult. And for multi-dimensional data (2D or 3D), the binding between work-items and data elements will make the situation worse.

B. Challenge II: Work-items Masking and Binding Switches

For applications like reduction, multiple rounds/passes are required to execute a single kernel, and not every work-item has to remain active in each round. Thus, programmers need to deactivate/mask work-items that are not used in the next round (see Figure 3(a) and 3(b)) to avoid unnecessary data updates. Furthermore, the bindings between work-items and data elements change even in a single kernel. In Figure 3(c), work-items $t_0$ and $t_1$ process data elements $d_0$ and $d_1$ in one round, but switch to update $d_1$ and $d_3$ in the following round, respectively. This binding switch makes code even more complex and confusing.

C. Challenge III: Inefficient Local Memory Organization

Local memory often works as temporal storage or plays the role of registers. In such cases, each work-item requires multiple data elements. The way that these elements are stored in local memory can have a significant impact on performance. Figure 4 shows two typical ways to organize the local memory space: Block and Cyclic. To avoid performance penalties (e.g., due to bank conflicts), programmers need to choose a proper way according to the access patterns of applications. Being aware of how data is organized and how accesses are mapped
to banks, programmers can avoid these penalties. However, understanding the banks scheme and mapping strategy of the local memory need a detailed code analysis, which is an error-prone and time-consuming process.

Summarizing the basic operations and challenges we presented above, we specify four high-level components of our API and their functionalities (CP and CH are short for ‘Component’ and ‘Challenge’, respectively):

1) GM → LM Operations: this component has to cover the operations needed to transparently load data from global memory to local memory (CP1 ∼ CH1).
2) LM → Registers Operations: this component has to provide programmers with simplified ways to address the move from local memory to registers (CP2 ∼ CH2).
3) Communication Operations: this component has to enable the users to make use of local memory as a synchronization and communication mechanism (CP3 ∼ CH3).
4) Local Memory Management Operations: this component allows programmers to efficiently organize and operate the local memory when it works as temporal storage or plays the role of the scarce registers (CP4 ∼ CH4).

IV. ELMO DESIGN

To satisfy the requirements (the four CPs) mentioned in Section III, we design ELMO as a middle layer between kernels and the basic operations for the local memory (see Figure 5). The basic idea is to keep the bindings between work-items and data elements transparent to users via a high-level API. The ELMO APIs consist of: (1) Block-Write Random-Read APIs (BWR), (2) APIs for Communications (COM), and (3) APIs for Local Memory Management (LMM).

BWR, proposed to address CP1 and CP2, allows the data to be loaded from global memory (into the local memory) in a block-wise way and used in arbitrary (or random) patterns presented in kernels. This API includes two operations: writing data from global memory into local memory (G2L), and reading data from local memory into registers (L2R). For G2L, users only need to give simple information such as the global/local memory addresses and the radius (see the model in Section V-A). After that, we enable the index conversion from global space to local space in L2R. Thus, BWR makes the process of moving data between local memory and global memory or registers transparent to the users.

This API can be widely used in real-world applications. First, BWR can be used for data sharing in applications where the data elements needed by one work-item overlap with the ones needed by its neighbors. Image convolution is a typical example of such an application. Second, the local memory can be used to explicitly convert a scattered access pattern to a regular (coalesced) pattern for read/write from/to global memory [9]. Thus, BWR is built to achieve high memory bandwidth even when the original memory access patterns are architecture-‘unfriendly’ (reversed or random memory access). Matrix transpose and cross-based aggregation in stereo matching [10] are representative examples of such applications.

COM, proposed to address CP3, encapsulates the complex communication and synchronization procedures that include multiple rounds to update data elements, and thus need work-item masking and binding switches. The encapsulation hides these confusing coding details from users.

Currently, the API mainly includes aggregation operations, such as reduction, prefix sum, and scan, but other communication operations like binomial reduction (often used in BinomialOption [9]) are to be added. The aggregation involves multiple passes to read/write data elements from/into memory, possibly in the form of irregular accesses. Thus, it is expected that local memory performs better on aggregations than global memory. Further, the aggregations usually require data communication (via work-item masking and binding switches) between work-items, which means that the registers cannot be used to replace local memory.

LMM, proposed to address CP4, aims to manage local memory space efficiently. The LMM API differs from the COM API in that the work-items within the same work-group process on their own space and do not have to communicate with each other via local memory. Thus, efficiently organizing and operating the local memory space becomes the main concern of LMM.

An example of a LMM operation is initialization, i.e., setting an element of a local memory region to a certain value (e.g., when computing histograms, memory has to be initialized to be zero). The operation should be straightforward, but programmers are likely to ignore the avoidance of bank-conflicts during initialization. Thus, we delegate this task to the LMM API to avoid the performance pitfall.
A different example of a LMM operation is relocation, needed to avoid the performance hits of register spilling. Register spilling occurs because OpenCL compilers give priority to allocating private memory for each work-item on register files (Figure 1), but these are limited in size. When an instance of a kernel consumes too many registers, register spilling occurs. The spilled data is usually transferred to global memory, which increases memory traffic and instruction count. Thus, local memory plays back up of registers via relocation.

To summarize, the front-end of ELMO comprises, by design, of three APIs that aim to enable the easy use of local memory. The design is user-oriented, decoupling the programmability issues from the performance ones (left for the back-end implementation).

V. ELMO IMPLEMENTATION

In this section, we provide implementations for each API. Specifically, we compare different implementations via micro-benchmarking, perform GPU-oriented optimizations, and give our preference on different application constraints.

A. BWR

By design, BWR consists of two separate steps: (1) G2L, loading the required data from global memory to local memory, (2) L2R, reading the data in the local memory when performing computation.

G2L When performing computation, each work-item needs to load data elements in the area of radius r, centered on its thread index (shown in Figure 6(a)). Note that r can be different between the top, bottom, left, and right halo data. When r = 0, i.e., no halo data, we use the 1-to-1 mode (between work-items and data elements) to load data, saturating the global memory bandwidth. However, when r > 0, the bandwidth of global memory may not be saturated, if the geometry of the input data block is not corresponding with that of a work-group. There are two ways we propose for this process (also shown in Figure 6(b), 6(c)): (1) reading data in a tile-by-tile fashion (TBT), or (2) loading the central data first, and then the halo data (FCTH).

![Fig. 6. G2L Model: the work-group is $4 \times 4$, and the radius is 1. Thus, each work-item (e.g., work-item 00) needs a $3 \times 3$ data block, and each work-group needs to load a $6 \times 6$ data block. When loading data into local memory, TBT needs 4 passes, numbered with 1, 2, 3, and 4 (the area outlined by dashed-line squares is given for illustrative purposes). FCTH first reads the central data, and then use 8 extra passes to load the halo data.](image)

From Figure 7, we see that FCTH performs better than TBT when $r \leq 16$. This is because TBT introduces extra branch overheads for a generic implementation. Further, when the dimension (width/height) of the local data block is not a multiple of that of a work-group (e.g., $r = 9$), TBT will lead to many more ‘small tiles’, wasting memory bandwidth. Thus, we prefer selecting the FCTH implementation when the radius is smaller than the dimension of a work-group, but we need to use TBT when this is not the case.

![Fig. 7. The data loading time from global memory to local memory and speedup (FCTH versus TBT). The data is obtained when the work-group is $16 \times 16$, and the input data is $2048 \times 2048$.](image)

L2R When performing computations using data elements in local memory, the key issue is to determine the correspondence $F$ between the index $(D_{gx}, D_{gy})$ of the data elements in the global data space and the index $(D_{lx}, D_{ly})$ of the data elements in the local data space within one work-group (shown in Equation 1).

$$F : (D_{gx}, D_{gy}) \rightarrow (D_{lx}, D_{ly}) \quad (1)$$

As we know,

$$D_{gx} = T_{gx} + \delta_x, \quad D_{gy} = T_{gy} + \delta_y,$$
$$D_{lx} = T_{lx} + \sigma_x, \quad D_{ly} = T_{ly} + \sigma_y,$$
$$\sigma_x = \delta_x + r, \quad \sigma_y = \delta_y + r,$$
$$T_{gx} \sim T_{lx}, \quad T_{gy} \sim T_{ly}. \quad (2)$$

From Equation 2, we can establish the correspondence $F$ ($\delta$ and $\sigma$ in the equations are implementation and iteration dependent parameters; $(T_{lx}, T_{ly}), (T_{gx}, T_{gy})$ are the local and global work-item index). When implementing the L2R API, we first make a conversion of data index from the global space to the local space, and then use the data in the local memory space.

B. COM

The implementation of the COM API is based on a generalization of [11], in which the authors present a segmented scan algorithm and its CUDA implementation. It consists of three steps: (1) intra-warp scan, (2) intra-block scan, and (3) global scan. We generalize this segmented algorithm to all the aggregation algorithms in the COM API’s implementations. Since the schedule units (warp from NVIDIA [8] and wavefront from AMD [9]) differ in size/width across vendors, we start with intra-block operations, and organize the algorithms as follows:
1) Loading the input array into local memory (some applications generate on-the-fly data as input).
2) Performing aggregations at the work-group-level (within one work-group).
3) Performing global aggregation on the results from all the work-groups.

For the moment, the supported aggregation operations in ELMO are reduction, prefix sum, and scan. As a proof of concept, we illustrate the reduction API and its implementation as follows:

**Reduction** needs to aggregate (sum, avg, max, min) the data to one final value. Each invocation of the kernel reduces the input array block to a single value within one work-group; it then writes this value to the output and reduces the partial results to a final result, which is sent to the host [9]. The reduction of each work-group is done in multiple passes. In the first pass, half of the work-items are active, and they update their values in local memory by aggregating the other half. This continues as shown in Figure 8.

![Figure 8](image)

Fig. 8. Reduction (sum): the circles represent work-items, the squares represent data elements in local memory, and the long bars separate work-items into different work-groups (4 work-items in each work-group).

The above-mentioned reduction maps one work-item to one data element. In practice, each work-item can perform reductions on multiple data elements (i.e., granularity coarsening). In Figure 9, we see that the granularity-aware implementation performs poorly for very small granularity due to more overheads from work-item creation, branches, and synchronization. The performance also decreases slightly when the granularity is too large because of less work-groups to hide latency. Therefore, we provide a parameter granularity in the API, for performance tuning.

![Figure 9](image)

Fig. 9. Full reduction time for different data size (51200-1638400) and granularity.

### C. LMM

The back-end of LMM has to optimize the organization and management of the local memory. The two operations - the memory initialization and relocation - are used to show the types of optimizations needed for LMM, their complexity and performance impacts.

**Initialization** Initializing the local memory is required when we perform statistics. For example, calculating a histogram needs to reset all bins prior to the computation itself. Initialization is independent of local memory organization (block or cyclic), i.e., there is no inherent binding between work-items and the local memory space. We compare two typical approaches to initialize the local memory: row-major (RMI) and column-major (CMI). For RMI, neighboring work-items perform the initialization on data elements in the same row, while CMI initializes the data elements in the column-major order (by neighboring work-items).

Table I shows the performance comparison of these two approaches. We see that RMI performs much better than CMI, with speedup ranging from 2.4× to 13.5×. This happens because CMI forces all the accesses into one memory bank, leading to serializing the writing operations, while RMI can successfully avoid it. Thus, we choose to use RMI in ELMO.

**TABLE I**

<table>
<thead>
<tr>
<th>N</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMI(ms)</td>
<td>0.35</td>
<td>1.15</td>
<td>4.44</td>
<td>11.54</td>
<td>45.32</td>
<td>269.89</td>
</tr>
<tr>
<td>RMI(ms)</td>
<td>0.15</td>
<td>0.22</td>
<td>0.45</td>
<td>0.90</td>
<td>3.41</td>
<td>19.99</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.40</td>
<td>5.12</td>
<td>9.85</td>
<td>12.81</td>
<td>13.27</td>
<td>13.50</td>
</tr>
</tbody>
</table>

**P2L** Using this API, we can store the likely to-be-spilled variables to local memory, rather than in private memory. The native kernels start with the cases that use private memory in the form of an array or a variable. Where register spilling could occur, we transform the usage of private memory to the usage of local memory (i.e., we relocate the variables from the private to the local memory). A comparison of the two implementations is shown in Figure 10.

![Figure 10](image)

The key issue to be solved here is index conversion from private space to local space. Once the way of organizing the local memory is determined (in Figure 4), we convert the index as follows:

\[
D_1 = \begin{cases} 
T_1 \cdot N + D_p & \text{if Block} \\
D_p \cdot S + T_1 & \text{if Cyclic}
\end{cases}
\]

(3)

where \(D_1, D_p, T_1\) represent local data index, private data index, and local index of a work-item, respectively; \(N\) is the number of variables required by one work-item, and \(S\) is the work-group size. When the spilled data is a variable, rather than an array (\(N = 1\) and \(D_p = 0\)), these two approaches will be one and the same.
Currently, we only provide users with the index conversion APIs. Users remain responsible for detecting the potential cases of register spilling (e.g., from the verbose information returned by compilers), but with the help of this API, users can be in control of the spilling and prevent expensive spills to global memory. For the future, an automated tool would be needed to detect the occurrence of register spilling, and relocate the to-be-spilled variables to local memory.

VI. EXPERIMENTAL EVALUATION

In this section, we present our experiments with ELMO focusing on the performance improvements it brings from two angles: (1) comparing the performance with native kernels (i.e., kernels without local memory or using local memory improperly) to evaluate our implementations and optimizations in ELMO, (2) comparing the performance with hand-tuned kernels further to see how ELMO performs and how far we can go in terms of performance.

A. Experimental Setup

All the experiments are performed on a NVIDIA Quadro5000 Fermi GPU. The card has Compute Capability 2.0 and consists of 352 cores divided among 11 multiprocessors. The number of 32-bit registers allocated to each multiprocessor is 32K, while the amount of local memory available per multiprocessor is 48K. We compile all the program with the OpenCL implementation from CUDA version 4.1 and GCC version 4.4.3.

Besides Reduction (RD) mentioned in Section V-B, we have implemented five more representative kernels in our experiments: Image Convolution (IC), Matrix Transpose (MT), Cost Aggregation (CA), Histograms (HT), and Marching Cubes (MC) using ELMO. Furthermore, we have applied ELMO to five applications from the AMD/NVIDIA SDKs, and compare ELMO’s performance to that of hand-tuned code.

B. Performance Comparison with Native Kernels

Image Convolution We implement the kernel in three different ways: (1) Native kernel without local memory (Native); (2) Optimized kernel using the BWR API in TBT mode (OPT-TBT); (3) Optimized kernel using the BWR API in FCTH mode (OPT-FCTH). In Figure 11, we see the performance for these three implementations. When \( r = 1 \), the optimized kernels perform worse than the native kernel (by around 32% and 16% for OPT-TBT and OPT-FCTH, respectively). This slowdown is due to the extra overhead of accessing local memory and the branches introduced in ELMO. The gain from data sharing in this application is offset by the overhead. Thereafter, the optimized kernels can yield significant performance improvement compared to the native implementation, with speedups of 1.3 \( \times \) -2.8 \( \times \) for OPT-TBT and 1.5 \( \times \) -3.1 \( \times \) for OPT-FCTH.

Matrix Transpose Since \( r = 0 \) (i.e., no halo data), we use the FCTH approach from ELMO for the optimized implementation. The results for different implementations and multiple data sizes are shown in Figure 12. We see that using ELMO will improve performance significantly, especially when the input data size is large. The native kernel (without local memory) violates the coalescing access constraints either on reading or on writing. While using ELMO, the coalesced access from global memory to local memory is ensured and operating the local memory itself has no coalescing constraints. When the input matrix size is small, the performance gap is minor, because the input data fits the L1/L2 caches on Quadro5000.
reading data elements from the local memory, bank-conflicts will occur. We remove the bank-conflicts by padding data, i.e., changing the row-size of the local memory (with an additional parameter in the BWR G2L API). Figure 12 shows the further performance improvement of the optimized kernel $OPT_{BCR}$ without bank-conflicts.

**Cost Aggregation** is a necessary step in local-based stereo matching of computer vision [12]. Cross-based cost aggregation [10] is a typical approach, in which performing computation on each pixel depends on an adaptive area of data elements around it, and the maximum radius is limited by a pre-defined $L$ value (see Figure 13(a)). This leads to un-coalesced global memory access with extremely low memory bandwidth. Thus, we use the BWR API to load all the data elements within the area of radius $L$ (more data elements than needed).

We use the four data sets (cones, teddy, tsukuba, and venus) from Middlebury [12] to evaluate the performance. The maximum limit $L$ is 17, which is larger than the dimension of a work-group, meaning that we can only use the TBT approach. Figure 13(b) shows the execution time for the two implementations ($Native$ versus $OPT$). We see that the optimized solver can achieve decent performance improvement, with speedup around $1.5 \times$. Although we load more data elements than what we need in this situation, we have achieved better performance using ELMO, due to the coalesced access from global memory to local memory.

![Data Area](image1)

(a) Data Area

![Execution time](image2)

(b) Execution time

Fig. 13. Cost Aggregation. (a) shows the shaded data elements are needed when performing cost aggregation for pixel $p$, with a pre-defined maximum radius of $L$. When using ELMO, we also load the un-shaded data elements besides the shaded ones. (b) shows the execution time and speedup on four datasets.

**Histograms** Calculating a histogram requires both local memory initialization and partial results summary. Figure 14 shows the results for the histogram implemented using ELMO-LMM, comparing the execution time of all four combinations of RMI/CMI initialization and block/cyclic organization. The optimization on initialization accounts for more than that on partial results summary, with an average speedup $1.46 \times$ versus $1.31 \times$. The combined optimizations can achieve a speedup of $1.27 \times$ to $2.96 \times$ compared with the native implementation.

![Histograms Execution Time](image3)

Fig. 14. Execution time of Histograms with different optimizations. The optimizations are in the format: Native=$\{CMI + Cyclic\}$, $OPT1=${$RMI + Cyclic$}, $OPT2=${$CMI + Block$}, $OPT3=${$RMI + Block$}.

**Marching Cubes** is a computer graphics algorithm for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field. One of the five kernels implemented in the NVIDIA SDK MC is called $generateTriangles$, which is used to calculate the flat surface normal for each triangle [8]. In the kernel, we allocate 16 $float 4$ data elements for each work-item to find the vertices where the surface intersects the cube, i.e., each work-item requires more than 64 registers, exceeding the capability of Quadro5000.

![Marching Cubes Execution Time](image4)

Fig. 15. Execution time and speedup of MC with maximum register limits.

We use ELMO’s P2L API and compare two implementations: PV - allocate all the variables directly on private space, and LM - allocate the array variable on local memory. The execution time for $generateTriangles$ when limiting the maximum number of registers per work-item is shown in Figure 15, for an input grid of $32 \times 32 \times 32$. The increase in number of registers limits results in performance improvement, especially from 16 through 24 to 32. Comparing to the register-only implementation, using local memory can significantly improve the performance (up to $2 \times$ faster). This is because the compiler spills data to the global memory space for the register-only implementation, which can be avoided by using ELMO.

C. Performance Comparison with Hand-tuned Kernels

To compare the performance with that of hand-tuned kernels, we select five equivalent kernels (i.e., kernels with the same algorithms) from AMD/NVIDIA SDKs, and apply ELMO to them. The results are shown in Table II (For CA, we cannot find a comparison reference from SDKs).
ELMO performs for different kernels. As for BoxFilter and Histograms, using communications are application-specific. Thus, future work is remaining applications, the patterns of data transformations and these, can be covered by ELMO (see Table IV). For the remaining applications, the patterns of data transformations and communications are application-specific. Thus, future work is to investigate how to abstract these operations to more generic local memory access patterns and include them in ELMO.

From Table II, we observe that the performance varies for different kernels. As for BoxFilter and Histograms, using ELMO performs $1.24 \times$ and $2.63 \times$ faster than the hand-crafted kernels. When looking into the hand-tuned code of BoxFilter, we notice that it uses an extended work-group (while we use the TBT approach of the ELMO BWR): the boundary work-items only participate in loading data from global memory to local memory, and remain idle for the other time-slices. The performance improvement of Histograms comes from the efficient initialization, i.e., the usage of RMI rather than CMI, enabled by ELMO.

For MatVecMul, however, ELMO performs 23% worse than the hand-tuned kernel. This is because the hand-tuned kernel uses warp-specific optimizations when performing reduction, which does not require any synchronization between two consecutive reduction passes. For the moment, we ignore these vendor-specific optimizations for portability on multiple and future platforms. Finally, we see that ELMO can achieve comparable performance with the hand-tuned versions of MarchingCubes and Transpose.

### VII. Discussion

In this section, we discuss the productivity and usability of ELMO. We also discuss the situations when ELMO is not suitable.

#### A. Productivity

To estimate the productivity to be gained by using ELMO, we assume that a typical user will use one API call instead of a number of lines of code (LOC) with the same functionality. Given that our back-end is an optimized, yet generic implementation of the API, it is fair to assume that an average programmer will use 75% ~ 100% of the lines of code from ELMO’s back-end, for a custom-made implementation of the same functionality. Therefore, for each ELMO API call, the code is shorter, on average, by 22 ~ 30 lines (see APIs and LOC in Table III). This means that using ELMO can not only simplify code writing, but avoid code bloating from non-computation related elements.

#### B. Usability

To get an idea of the usability potential of ELMO, we have investigated the AMD/NVIDIA SDK applications. In total, we have found 30 applications that use local memory. Out of these, 20 can be covered by ELMO (see Table IV). For the remaining applications, the patterns of data transformations and communications are application-specific. Thus, future work is needed to investigate how to abstract these operations to more generic local memory access patterns and include them in ELMO.

### Table II

Performance comparison with hand-tuned kernels.

<table>
<thead>
<tr>
<th>Applications</th>
<th>EQ. Kernels</th>
<th>SDKs</th>
<th>DataSize</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoxFilter</td>
<td>IC</td>
<td>NVIDIA</td>
<td>1024x1024</td>
<td>1.24</td>
</tr>
<tr>
<td>Transpose</td>
<td>MI</td>
<td>NVIDIA</td>
<td>2048x2048</td>
<td>0.97</td>
</tr>
<tr>
<td>MatVecMul</td>
<td>CA</td>
<td>NVIDIA</td>
<td>1100x1100</td>
<td>0.72</td>
</tr>
<tr>
<td>Histograms</td>
<td>HI</td>
<td>AMD</td>
<td>2048x2048</td>
<td>2.63</td>
</tr>
<tr>
<td>MarchingCubes</td>
<td>MC</td>
<td>NVIDIA</td>
<td>32x32x32</td>
<td>1.04</td>
</tr>
</tbody>
</table>

### Table III

Overview of ELMO and its LOC.

<table>
<thead>
<tr>
<th>APIs</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWR</td>
<td>BoxFilter, ConvolutionPerpendicular, FDTD3d, MatrixMul, MedianFilter, NBody, Particles, RecursiveGaussian, SobelFilter, MatrixTranspose, LIDecomposition, QuadRandomSequence</td>
</tr>
<tr>
<td>COM</td>
<td>HiddenMarkovModel, MatrixVecMul, Reduction, ScanLargeArrays, PrefixSum</td>
</tr>
<tr>
<td>LMM</td>
<td>Histogram, MarchingCubes, URNG</td>
</tr>
<tr>
<td>Non-yet</td>
<td>DCTSSK, DXTCompression, RadixSort, SortingNetworks, Triadagonal, AESEncryptDecrypt, BinomialOption, DtwHaar1D, FFT, GaussianNoise</td>
</tr>
</tbody>
</table>

### Table IV

Applications covered (already and not-yet) by ELMO.

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWR</td>
</tr>
<tr>
<td>COM</td>
</tr>
<tr>
<td>LMM</td>
</tr>
<tr>
<td>Non-yet</td>
</tr>
</tbody>
</table>

### Table V

The execution time (in ms) of NBody with different bodies.

<table>
<thead>
<tr>
<th>#Bodies</th>
<th>10240</th>
<th>20480</th>
<th>40960</th>
<th>81920</th>
<th>163840</th>
<th>327680</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notice</td>
<td>1.81</td>
<td>0.75</td>
<td>25.32</td>
<td>98.42</td>
<td>389.76</td>
<td>1580.24</td>
</tr>
<tr>
<td>slowdown</td>
<td>1.14</td>
<td>1.07</td>
<td>1.13</td>
<td>1.11</td>
<td>1.12</td>
<td>1.11</td>
</tr>
</tbody>
</table>
Finally, we note that (at the time of writing), OpenCL for Multicore CPUs will map local memory onto global memory. All memory objects are cached by the hardware and explicitly using local memory for caching will most likely add unnecessary overheads [13]. Thus, more research is required in implementing a CPU-friendly back-end. Until then, ELMO is recommended for architectures equipped with a separate, fast local memory.

VIII. RELATED WORK

In this section, we discuss prior work on tools, compilers, and optimization techniques of using local memory. Baskaran et al. [1] develop an approach to effective automatic data management for on-chip memories, including creation of buffers in on-chip memories for holding the needed data elements, determination of array access functions, and generation of code that moves data between slow off-chip memory and fast local memories. In [14], Yang et al. propose a GPGPU compiler, which converts the un-coalesced accesses to coalesced ones, and enhances the data reuse. Because of our analysis of the challenges of using local memory, we present broader and more generic uses of local memory, besides their proposed data reuse and data layout changing.

In [15], Bauer et al. present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memory of NVIDIA GPUs. The driving force of CudaDMA is to emulate the use of asynchronous hardware DMA engines for GPUs at a software level. However, there is no DMA hardware on GPUs, which makes the asynchronous approach less fascinating. Further, the CudaDMA uses two classes of warps: DMA warps and Compute warps, taking charge of data movement and computation, respectively. This warp-specialization implementation introduces code bloat, as we can see in the paper. In contrast, we build our ELMO based on multiple models and keep the native way of using local memory, allowing our APIs to be easy-to-use and user-friendly.

Research on local memory optimization techniques is yet another interesting related topic. For example, in [16], Moazeni et al. present a memory reuse technique to minimize the use of local memory space. This is to address the concern that an incremental increase in the usage of local memory per thread can result in a substantial decrease in performance. In [17], Ren et al. propose a framework for automatically tuning applications to machines with software-managed memory hierarchies. Such techniques could be used to further optimize ELMO’s back-ends.

Overall, our related work survey shows that (1) ELMO’s design is novel and could provide users an alternative way of using local memory in OpenCL, and (2) the back-end can be further extended with more complex, yet fairly portable optimizations.

IX. CONCLUSIONS

On multiprocessors with explicitly managed memory hierarchies, programmers have the responsibility of moving data in and out of the local memory for high performance. This task can be complex and error-prone even for expert programmers. Thus, we propose the ELMO API to improve productivity while preserving high performance. Addressing the challenges of (1) geometry mismatch, (2) work-items masking and binding switches, and (3) inefficient local memory organization, the API presents a user-friendly interface and covers diverse using scenarios. Our experimental results show that ELMO can improve performance by up to 3.7× on NVIDIA Quadro5000. Even when compared with hand-tuned code for using local memory, ELMO can deliver matching performance and does not hinder other types of hand-tuning.

For the front-end, ELMO still needs extensions to cover more access patterns from real-world applications. For the back-end, we plan to investigate optimizations and performance tuning on more OpenCL-compliant platforms. Our long-term goal is to develop automated tools that, based on ELMO, will optimize the use of local memory.

ACKNOWLEDGMENT

We thank our colleague Alexander S. van Amersfoort for his advice and expertise during several fruitful discussions. This work is partially funded by CSC (China Scholarship Council), and the National Natural Science Foundation of China under Grant No.61103014.

REFERENCES