View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Scientific Programming 22 (2014) 239–257
DOI 10.3233/SPR-140390
IOS Press


# Aristotle: A performance impact indicator for the OpenCL kernels using local memory

Jianbin Fang [a,*], Henk Sips [a] and Ana Lucia Varbanescu [b]
[a] *Delft University of Technology, Delft, The Netherlands*
*E-mails: {j.fang, h.j.sips}@tudelft.nl*
[b] *University of Amsterdam, Amsterdam, The Netherlands*
*E-mail: a.l.varbanescu@uva.nl*

**Abstract.** Due to the increasing complexity of multi/many-core architectures (with their mix of caches and scratch-pad memories) and applications (with different memory access patterns), the performance of many workloads becomes increasingly variable. In this work, we address one of the main causes for this performance variability: the efficiency of the memory system. Specifically, based on an empirical evaluation driven by memory access patterns, we qualify and partially quantify the performance impact of using local memory in multi/many-core processors. To do so, we systematically describe memory access patterns (MAPs) in an application-agnostic manner. Next, for each identified MAP, we use OpenCL (for portability reasons) to generate two microbenchmarks: a "naive" version (without local memory) and "an optimized" version (using local memory). We then evaluate both of them on typically used multi-core and many-core platforms, and we log their performance. What we eventually obtain is a local memory performance database, indexed by various MAPs and platforms. Further, we propose a set of composing rules for multiple MAPs. Thus, we can get an indicator of whether using local memory is beneficial in the presence of multiple memory access patterns. This indication can be used to either avoid the hassle of implementing optimizations with too little gain or, alternatively, give a rough prediction of the performance gain.

Keywords: Micro-benchmarking, local memory, memory access pattern, OpenCL

## 1. Introduction

In the last few years, multi/many-core processors have been becoming extremely popular. With the increasing number of processing cores, the ratio of CPU to memory speed is growing rapidly. To exploit the full benefits of the increasing number of processing cores, architects need to ensure that memory bandwidth are optimized. Utilizing a cache hierarchy, as well as increasing the cache size, are traditional approaches to alleviate the memory bottleneck [3].

Alternatively, modern multi/many-core processors like GPUs use programmer-managed *scratch-pad memories (SPM)*. Studies have shown that scratch-pad memories use 34% lesser area and consume 40% less power than a cache of the same capacity [27]. Since the on-chip cache typically consumes 25–50% of the processor's area and energy, these savings are significant. Like caches, scratch-pad memories are situated on-chip and are much faster than the off-chip memories. Therefore, proper use of scratch-pad memories often leads to a higher effective memory bandwidth and a better overall performance.

*OpenCL (Open Computing Language)* [15], the standard proposed by the Khronos Group for programming many-cores, recognizes SPMs under the name of *local memory* in its conceptual device architecture.[1] Eager to be part of the development and deployment of the common programming model for many-cores, many vendors have implemented OpenCL and local memory on top of their hardware and software stacks. For example, NVIDIA maps local memory onto the on-chip SPM, while the cache-only processors[2] such as the multicore CPUs map it to the off-chip memory [9].

Due to architectural disparities and, in particular, the differences in implementing local memory, programmers often use the *trial-and-error* approach to enable

[*] Corresponding author. E-mail: j.fang@tudelft.nl.

[1] NVIDIA uses the term 'shared memory', while AMD calls it 'local data store'. In this paper, we use the OpenCL name 'local memory' [15].

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

1058-9244/14/$27.50 © 2014 – IOS Press and the authors. All rights reserved

local memory and evaluate its efficiency: taking a naive kernel, they translate the code into an “optimized” version that uses local memory and then measure its impact. This is a time-consuming process, as programmers have to address, in their OpenCL code, challenges like (1) geometry mismatches, (2) work-items[3] masking and binding switches, and (3) inefficient local memory organization [6]. Similarly, for architectures where using local memory is not recommended, programming effort is often spent on removing the code related to local memory for improved performance. We argue that solving these problems requires a lot of effort to be spent on non-computational and non-functional details of kernels, which hinders productivity. Therefore, we propose a solution to quantify the performance impact of using local memory *before* implementing it. Our analyzer will help sparing a lot of useless programming effort.

Despite common belief [1,11,22], the impact of local memory usage on performance is not easy to determine. For example, *data reuse* is a commonly recognized source of performance gains of using local memory [8,12,19,26]. However, data reuse and local memory are not always correlated: data reuse does not automatically lead to a higher local memory efficiency (see Section 2.2.1), nor does the lack of data reuse mean lack of performance improvement (see Section 2.2.2). Furthermore, in the case of CPUs, the off-chip placement of the local memory makes programmers choose not to use it [11], but properly using it can lead to performance improvement (see Section 2.2.3).

In this work, *we address the issue of performance unpredictability when using local memory in a two-stage approach: quantification and composition*. For quantification, we develop a benchmark-based approach to quantify the performance impacts of using local memory for 33 memory access patterns (MAPs) in isolation. For each MAP, we generate two types of benchmarks: with and without using local memory. We empirically evaluate these benchmarks on typically used platforms, and record the achieved performance in a *performance database*. In practice, we can obtain the performance benefits of using local memory for a single MAP by querying the database. For composition, we present a set of rules (empirically validated) to determine whether to use local memory or not in the presence of multiple MAPs. The database plus the composing rules will produce an indicator of whether to use local memory for a given application. We name the approach, including the code generator and validator, as *Aristotle*.

[3]In the context, we use ‘work-item’ and ‘thread’ interchangeably.

To summarize, we make the following contributions:

- We formalize the benchmark design space and develop a code generator which helps applying our approach on any OpenCL-compliant platform.
- We evaluate the performance impacts of using local memory on a broad category of processors and generate a comprehensive and representative database.
- We design and validate a set of composing rules to determine whether to use local memory in the presence of multiple MAPs.

The paper is organized as follows: We list three counter-intuitive observations of using local memory in Section 2. Our approach is presented in Section 3. We extend a mathematical model to describe memory access patterns and derive a set of MAPs in Section 4. We explore the design space of using local memory and produce benchmarks using a code generator in Section 5. We generate a performance database by running the microbenchmarks on seven typically used platforms in Section 6. In Sections 7 and 8, we propose and validate a set of composing rules in the presence of multiple MAPs. We present related work in Section 9 and we summarize our findings in Section 10.

## 2. Background and motivation

### 2.1. OpenCL and local memory

*OpenCL* is a relatively new standard for parallel programming of heterogeneous systems [15]. An OpenCL program has two parts: *kernels* that execute on one or more *OpenCL devices* (typically accelerators such as GPUs) shown in Fig. 1, 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 onto devices for execution, an index space of *work-items* (instances of the kernel) is defined. A work-item is typically executed on a *processing element (PE)* of the device. Further, work-items are organized into *work-groups*, which run on *computer units (CU)* in a lock-step fashion.

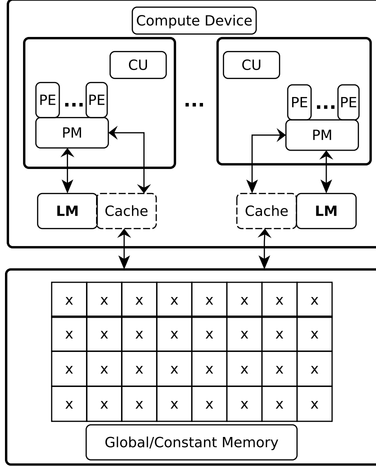



Fig. 1. Conceptual device architecture with processing elements (PE), compute units (CU). The memory regions include private memory, caches (architecture-related), local memory (LM), and global/constant memory. The host is not shown.

Each work-item has its own *private memory* space, and can share data via *local memory* with the other work-items in the same work-group. All work-items can read/write the *global memory*. Figure 1 shows the OpenCL device architecture with local memory [15]. To achieve a high bandwidth, local memory is divided into equally-sized memory *banks*, which are organized in such a way that successive words are assigned to successive banks, i.e., interleaved. A *bank conflict* occurs when two or more work-items access different words in the same bank [1,22]. 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.

## 2.2. Three observations

Our work is based on the observation that local memory, although perceived as a guarantee of performance gain, does not always behave so. In this section, we give a more detailed analysis of three types of such behaviors/observations.

### 2.2.1. Data reuse $\neq$ performance improvement

The occurrence of data reuse is a widely used criterion of moving data from global memory to local memory. However, this statement does not always hold. Table 1 shows the memory bandwidth when running NBody [23] on NVIDIA GTX580. We see that although the input data elements are shared by all the threads for NBody, using local memory performs worse than not using it (by around 20%). The performance loss is due to the fact that GTX580 has caches (L1 and L2) that make better use of data sharing than the local memory. Specifically, local memory enables data sharing among work-items within one work-group, while the L1 cache can identify the data sharing within one work-group, and the L2 cache will enable global data sharing on the input data (i.e., among work-groups as well). Additionally, using local memory introduces extra overheads for data movement operations in and out of local memory. Therefore, the caches may "cancel" the performance gains of using local memory.

### 2.2.2. No data reuse $\neq$ performance loss

Let us consider data movements between local memory and global memory. Suppose we have $N$ compute units and the bandwidth of local memory access is $W_l$. An application requires $D$ data elements to be moved when using global memory only (with a bandwidth of $W_g$), and $D'$ data elements to be moved from global memory to local memory (with a bandwidth of $W_g'$). We compute the time of data movement without ($T_{w/o}$) in Eq. (1) and with local memory ($T_{w/i}$) in Eq. (2). We see that performance improvement of using local memory comes from two factors: either the decrease of data amount ($D' < D$), and/or the increase of global memory bandwidth ($W_g' > W_g$). Thus, considering data reuse as a must for local memory performance gain is incorrect and will lead to missed opportunities for local memory usage. Taking a straightforward approach for a Matrix Transpose on GPUs for example, the implementation will violate the coalesced constraints on the global memory access. Using local memory, we can ensure coalescing for both input and output memory access, and thus improve the bandwidth:

$$T_{w/o} = \frac{D}{W_g}, \tag{1}$$

$$T_{w/i} = \begin{cases} \dfrac{D}{W_l} + \dfrac{D'}{W_g'}, & N = 1, \\ \max\left(\dfrac{D}{W_l}, \dfrac{D'}{W_g'}\right) = \dfrac{D'}{W_g'}, & N > 1. \end{cases} \tag{2}$$

### 2.2.3. Local memory use on CPUs $\neq$ performance loss

At the moment of writing, local memory is allocated within the main memory space of the CPUs (global memory in Fig. 1). Thus, it is not recommended to use local memory on CPUs [11]. However, we have found

Table 1
Memory bandwidth (GB/s) of NBody where the local memory is allocated dynamically

| | 64 × 64 | 128 × 128 | 256 × 256 | 512 × 512 | 1024 × 1024 |
|---|---|---|---|---|---|
| $LM_{w/o}$ | 613.50 | 636.43 | 646.06 | 616.42 | 589.95 |
| $LM_{w/i}$ | 512.44 | 495.28 | 516.04 | 518.61 | 520.64 |
| Loss (%) | 16.47 | 22.18 | 20.13 | 15.87 | 11.75 |

*Notes*: $LM_{w/o}$ represents the naive kernel and $LM_{w/i}$ represents the kernel using local memory. We use five datasets each with a different matrix/input size.

Table 2
Memory bandwidth (GB/s) of convolution with and without local memory for six datasets

| | 64 × 64 | 128 × 128 | 256 × 256 | 512 × 512 | 1024 × 1024 | 2048 × 2048 |
|---|---|---|---|---|---|---|
| $LM_{w/o}$ | 6.81 | 7.77 | 7.81 | 8.06 | 8.13 | 8.15 |
| $LM_{w/i}$ | 12.23 | 13.81 | 14.08 | 14.56 | 14.70 | 14.56 |
| Speedup | 1.80 | 1.78 | 1.80 | 1.80 | 1.81 | 1.79 |

that this does not always hold. Table 2 shows the memory bandwidth of a convolution kernel on Intel Xeon E5620 (a dual-socket 4-core processor). We see that using local memory delivers better performance than not using it (around 2× faster). Using local memory on CPUs introduces extra overheads, but it also changes the usage of caches and allows compilers to do specific optimizations for data placed by the users in local memory.

To summarize, these three counter-intuitive observations show that using local memory makes it difficult to predict the performance gain and thus the performance unpredictability. Further, our analysis indicates that the unpredictability results from the diversity of architectures/processors and applications.

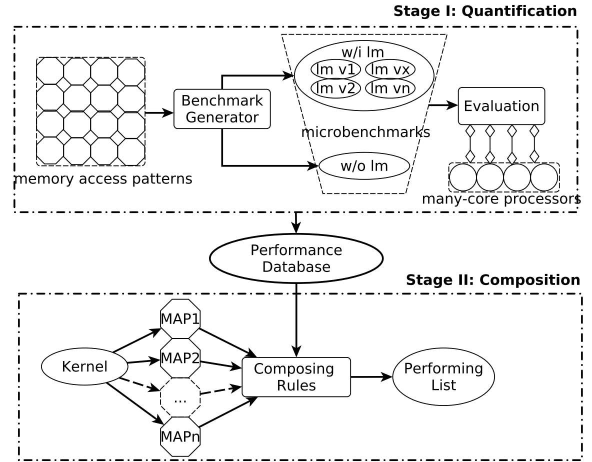



Fig. 2. Aristotle overview.

## 3. The design of Aristotle

Based on all these observations (Section 2), we believe that a better understanding of the cases when local memory is useful, and better quantifying its usefulness are equally required. Thus, we propose a hybrid approach to tackle this issue: use MAP modelling to generate microbenchmarks, and use traditional performance measurement to quantify local memory usefulness.

Figure 2 shows the Aristotle framework. For all memory access patterns, we generate 2 benchmark kernels: one without local memory, and the other one with local memory. Then we evaluate the benchmarks on typically used many-core processors and generate a performance database. Given a kernel, we identify the MAPs embedded in it and use the composing rules to generate a performing list of whether or not to use local memory on each MAP.

Note that our benchmarks start with memory access patterns (MAPs), which we consider to be models of the input kernels. Our goal is to evaluate the benchmarks empirically, giving accurate information on the benefit of using local memory. Thus, given an application MAP and a platform, a simple query in our database can show how using local memory impacts the application performance. Furthermore, the memory access patterns can be manually identified from input kernels [13], or automatically abstracted during runtime [24], and are, for now, outside the scope of this work.

## 4. MAP description

We express a MAP as a *memory access sequence*, which allows us to represent discrete memory refer-

ences and loops. Our approach is based on the notation in [13,17]. We make use of a similar notation, which enables us to study memory access patterns systematically. To keep the number of analyzed MAPs under control, we rewrite the formulation such that we clearly separate the inter-thread and intra-thread parallelism, and we focus on five categories of patterns. Specifically, we assume a 2D thread configuration $(t_x, t_y)$, for which we investigate the resulting inter-thread access patterns, and five different intra-thread access patterns, to match the most important MAPs found in real-life applications. Using these limitations, we are able to fully analyze a set of MAPs that are intuitive and cover a large set of real-life applications.

### 4.1. The notation

According to [13], a memory access sequence $\vec{s}$ can be expressed as a combination of a memory access matrix, **M**, an iteration vector, $\vec{i}$, and an offset vector, $\vec{o}$. The dependency is presented in Eq. (3). Note that this notation is applicable to loop nests of arbitrary depth, and depending on the mapping of these iterations on the threads space, the memory access matrix will cover both the inter- and intra-thread memory access patterns.

$$\vec{s} = \mathbf{M}\vec{i} + \vec{o}, \tag{3}$$

$$\vec{s} = e\vec{MAP} + i\vec{MAP} = \mathbf{M}\vec{tid} + i\vec{MAP}. \tag{4}$$

We have adapted this notation to express our specific range of MAPs – see Eq. (4). In this new notation, we have clearly separated the inter-thread ($e\vec{MAP}$) and intra-thread ($i\vec{MAP}$) components. Intuitively, eMAP generates a base access index for each thread, while *iMAP* provides an offset which represents the distance from the base address. We focus on 2D thread organization: **M** becomes a $2 \times 2$ mapping matrix of the threads ($\vec{tid}$) to the data. The $i\vec{MAP}$ component is a vector representation of the intra-thread access pattern. We further rewrite Eq. (4) to Eq. (5), and we use this form to exhaustively generate our benchmarks:

$$\vec{s} = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} iMAP_0 \\ iMAP_1 \end{bmatrix}. \tag{5}$$

### 4.2. eMAP

When $M_{00}, M_{01}, M_{10}, M_{11} \in \{0,1\}$, we generate 16 cases of eMAP (shown in Fig. 3). As we have mentioned, eMAP encodes the base index of the memory references for each thread. For example, Fig. 4 shows the base index of eMAP-14 for each thread. We assume a $8 \times 8$ workgroup, and a dataset of (at least) $15 \times 8$; for simplicity, in this example, we consider $i\vec{MAP} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. In this case, consecutive work-items in the $x$-dimension will access contiguous data elements in the horizontal direction; consecutive work-items in the $y$-dimension will access the elements on the diagonal line. Thus, the base index of each thread is located within the shaded area (Fig. 4(a)).

When $M_{00}, M_{01}, M_{10}, M_{11} \notin \{0,1\}$, the eMAPs become more complex. When $M_{00} = 2$, we see (Fig. 4(b)) 'gaps' between rows due to the larger stride, compared with eMAP-14. We can imagine that any non-unit stride will introduce such 'gaps'. For now, our work only considers $[0,1]$ cases (Fig. 3). We believe the extension to larger strides will not bring changes to our methodology. However, it will lead to cases very

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(01)} \quad \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(02)} \quad \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(03)} \quad \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(04)}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(05)} \quad \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(06)} \quad \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(07)} \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(08)}$$

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(09)} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(10)} \quad \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(11)} \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(12)}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(13)} \quad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(14)} \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(15)} \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} t_y \\ t_x \end{bmatrix} \text{(16)}$$

Fig. 3. eMAP cases (numbered 01 to 16).

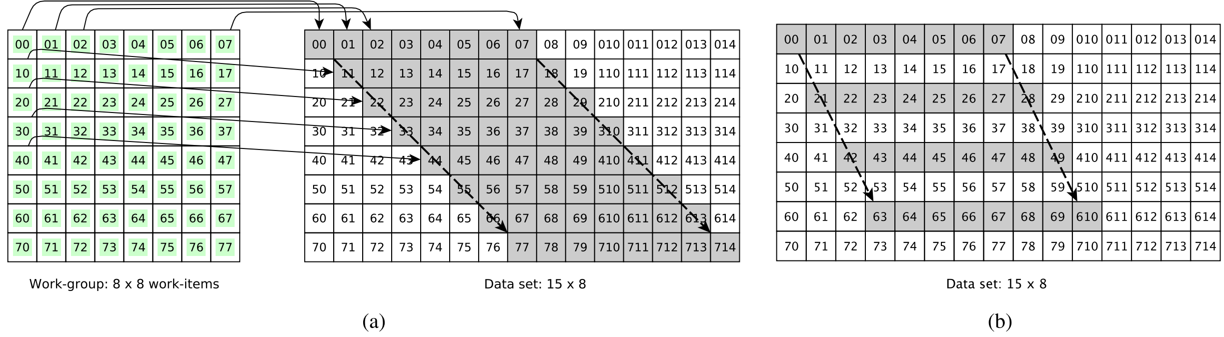



Fig. 4. Base index example: (a) eMAP-14: the shaded elements are the ones accessed by the whole $8 \times 8$ workgroup; the arrows indicate (some of) the one-to-one relations between threads and data items; (b) the base index in the data structure when $M_{00} = 2$ (only show the first four rows). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)

rarely seen in real-life applications and a large increase in the experimentation time.

### 4.3. *iMAP*

*iMAP* captures the memory access patterns of a single thread, i.e., the way one thread accesses data elements. We have identified five typical iMAPs from real-life applications [6] – namely, Single, Row, Column, Block and Neighbor – and briefly describe them:

- *Single* (1): each thread accesses one data element indexed by its base index.
- *Row* (2): each thread references a row of data elements within the row indexed by its base.
- *Column* (3): each thread accesses a column of data elements within the column indexed by its base index.
- *Block* (4): each thread accesses a block of data elements within the block centered at the base index and sized $((2R_x + 1) \times (2R_y + 1))$.
- *Neighbor* (5): each thread accesses the data elements lying at the base index and its four (or more) neighbors.

The *iMAP* representations are listed as follows, where $W$ and $H$ represents the width and height of the input matrix, respectively; $(R_x, R_y)$ is the radius of a block:

Single: $$\mathit{iMAP} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\},$$

Row: $$\mathit{iMAP} = \left\{ \begin{bmatrix} 0 \\ i \end{bmatrix} \;\middle|\; 0 \leqslant i < W, i \in N \right\},$$

Column:
$$\mathit{iMAP} = \left\{ \begin{bmatrix} j \\ 0 \end{bmatrix} \;\middle|\; 0 \leqslant j < H, j \in N \right\},$$

Block:
$$\mathit{iMAP} = \left\{ \begin{bmatrix} j \\ i \end{bmatrix} \;\middle|\; -R_x \leqslant i \leqslant R_x, i \in N; \; -R_y \leqslant j \leqslant R_y, j \in N \right\},$$

Neighbor:
$$\mathit{iMAP} = \left\{ \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}.$$

### 4.4. *MAP = eMAP + iMAP*

Once eMAP and iMAP are specified, we get 80 ($16 \times 5$) memory access patterns (MAPs), and hence need to generate and evaluate 80 microbenchmarks. The name of each MAP is a concatenation of the iMAP and eMAP numbers. For example, MAP-407 is a combination of iMAP-4 (Block) and eMAP-07. In the remainder of this paper, we also group MAPs by their iMAP name, having Single MAPs (the MAPs that have the "Single" iMAP), and similarly Row MAPs, Column MAPs, Block MAPs and Neighbor MAPs.

When analyzing our 80 MAPs, we find that some combinations of eMAP and iMAP are either under-specified (resulting in non-interesting cases) or over-specified (resulting in contradictory definitions). Take for example MAP-101, in which each thread should access one element (according to the iMAP), but due to the eMAP (01), all threads end up accessing the

same element (i.e., the $(0, 0)$ element from the dataset), an uninteresting case – i.e., an *underspecified MAP*. MAP-206 is also *underspecified*: all threads end up accessing the same row (i.e., row 0 from the dataset). On the other hand, MAP-216 is an *overspecified MAP*, as the eMAP and iMAP specify contradictory rules for accessing the same elements.

We generalize the classes of compatible eMAPs for each iMAP as follows:

- *Single*: **M** should have at least one '1' per row.
- *Row*: **M** should have no '1' on the bottom row, and at least one '1' on the top row.
- *Column*: **M** should have no '1' on the top row, and at least one '1' on the bottom row.
- *Block*: similar to (1).
- *Neighbor*: similar to (1).

After removing the under/overspecified MAPs, only 33 MAPs remain valid, and are listed in Table 3. Note that, in the paper, we do not take the random memory access into account since we assume that local memory is mainly suitable for applications with specific memory access patterns.

We note that this approach is, so far, application-agnostic. In other words, we attempt to generate all possible MAPs for our representation and evaluate their local memory impacts. Thus, our database is generic and fully reusable by any application.

Table 3
The memory access patterns

| | Single (1) | Row (2) | Column (3) | Block (4) | Neighbor (5) |
|---|---|---|---|---|---|
| **01** | – | – | – | – | – |
| **02** | – | – | 302 | – | – |
| **03** | – | – | 303 | – | – |
| **04** | – | 204 | – | – | – |
| **05** | – | 205 | – | – | – |
| **06** | – | – | 306 | – | – |
| **07** | 107 | – | – | 407 | 507 |
| **08** | 108 | – | – | 408 | 508 |
| **09** | 109 | – | – | 409 | 509 |
| **10** | 110 | – | – | 410 | 510 |
| **11** | – | 211 | – | – | – |
| **12** | 112 | – | – | 412 | 512 |
| **13** | 113 | – | – | 413 | 513 |
| **14** | 114 | – | – | 414 | 514 |
| **15** | 115 | – | – | 415 | 515 |
| **16** | 116 | – | – | 416 | 516 |

*Note*: '–' represents an impossible MAP – either under or overspecified.

## 5. Design space exploration and code generation

### 5.1. Exploring design space

When generating benchmarks (for a MAP) with local memory, we need to consider the issues of *local space allocation*, *local data staging* and *local memory access*.

#### 5.1.1. Local space allocation

Regarding the size of local space, we propose two approaches: the *min-approach* and the *max-approach* [5]. The min-approach allocates a right-sized space of local memory to hold the necessary data elements with none or very few wasted cells, while the max-approach allocates a large enough space according to the shape of a work-group. We demonstrate how these two approaches work for MAP-407 in Fig. 5, where $R_x = R_y = 1$ and thus each thread needs a $3 \times 3$ data block. Using the min-approach consumes less local memory (Fig. 5(b)), and may enable more work-groups active. Nevertheless, when using the min-approach, programmers need to perform work-item binding and data element shuffling according to specific memory access patterns. By comparison, the max-approach is easier for implementation (e.g., from a script). In this work, we implement both the max-approach and the min-approach, and we compare their performance in Section 6.3.1. Because we know the size of local space in advance, we use the static allocation approach (i.e., allocating local memory in the kernel).

When using the max-approach, we calculate the size of local space as follows. Each MAP has two parts – eMAP and iMAP, and the size of local space `Range` is determined by these two factors. The eMAP part specifies the `Base` (the area outlined by the dashed-line rectangle shown in Fig. 5(c)) and the iMAP part specifies the `Border`. Suppose the `Base` is of size $w \times h$, and $w$, $h$ is calculated as follows ($WG_x \times WG_y$ represents work-group size):

$$w = \begin{cases} WG_x & (M_{00} \oplus M_{01} = 1), \\ 2 \times WG_x & (M_{00} \wedge M_{01} = 1), \end{cases}$$

$$h = \begin{cases} WG_y & (M_{10} \oplus M_{11} = 1), \\ 2 \times WG_y & (M_{10} \wedge M_{11} = 1). \end{cases}$$

We can then calculate the `Range` covered by a work-group as $(w + 2 \times R_x) \times (h + 2 \times R_y)$. Furthermore, the use of the min-approach depends on the MAPs and thus it is a MAP-dependent optimization.

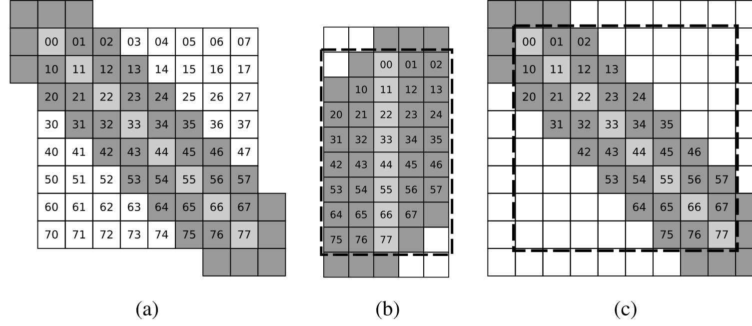



Fig. 5. Two approaches to hold data elements using local memory for MAP-407: (a) data elements in global memory space for a work-group of $8 \times 8$ (only the shaded cells need to be transferred into local memory), (b) data elements in local memory space using the min-approach (occupying 50 local memory cells), (c) data elements in local memory space using the max-approach (occupying 100 local memory cells).

### 5.1.2. *Local data staging*

After allocating the required local memory, we need to stage data in the local space specified by the `Range`. Note that this process is independent of how data is used (to be mentioned in Section 5.1.3), which provides a large degree of freedom to optimize the data staging process. In [6], we have proposed the `FCTH` (i.e., loading the base data first, and then the border data) and `TBT` (i.e., reading data in a tile-by-tile fashion) to stage the local data and shown that `FCTH` gives us a better performance. Hence, in this work, we use the `FCTH` approach.

### 5.1.3. *Local data access*

Compared with accessing the data in global space, the key issue of accessing local data is the `index space conversion`. Specifically, we need to use the `local` thread index instead of `global` thread index while keeping the logic of using global memory. In addition, to ensure that the work-items within a work-group efficiently reference the data elements in local space, we need to avoid bank conflicts, i.e., to force the access requirements from multiple work-items of a work-group fall into different banks. By using data padding, we remove bank-conflicts from the generated microbenchmarks.

### 5.2. *Code generator*

Our code generator consists of two templates: host code and kernel code. The engine of the host code creates a driver that allocates/deallocates global space, initializes the data space, transfers data between the host and the device and launches kernels. It also has a module of time keeping and results validation. With regard to the kernel code, each microbenchmark of using local memory includes three major steps: statically allocate local memory space, load data elements into local memory, and use them. We found that different iMAPs differ in their code generators. Thus, we have developed a different code generating engine based on the iMAPs. Figure 6 shows the kernel template for the Block iMAP.

```
1  __kernel void CG(@type in[], @type out[], int cdim, int rdim){
2      int tgx = get_global_id(0);
3      int tgy = get_global_id(1);
4      int tlx = get_local_id(0);
5      int tly = get_local_id(1);
6      int wgx = get_group_id(0);
7      int wgy = get_group_id(1);
8      @bxy  // Get the base index of a group
9      @lmAlc // Step 1: allocte local memory space
10     @varDec
11     // Step 2: load data from GM to LM
12     @lmLoad
13     barrier(CLK_LOCAL_MEM_FENCE);
14     // Step 3: use the data elements in LM
15     @lmUse
16     barrier(CLK_LOCAL_MEM_FENCE);
17     // output
18     out[tgy*cdim+tgx] = @lmOut;
19     return ;
20 }
```

Fig. 6. A code template to generate kernels in OpenCL (*@type* represents the used data type, *@bxy* represents the base data index for each group which is MAP-dependent, *@lmAlc* is the name-holder for local space allocation, *@varDec* declares temporal variables, *@lmLoad* is the name-holder of loading data into local memory and *@lmUse* is that of how to use it, and *@lmOut* is the final returned results).

Taking MAP-407 (Fig. 5) for example, we show the three steps in detail.

**Step 1** (Allocating local memory (*@lmAlc*)). We use the approaches mentioned in Section 5.1.1 to calculate the local size and allocate the local space. For MAP-407 (shown in Fig. 5), the min-approach and max-approach need different amounts of local space. The

local space size is calculated as $(WG + 2 \times R) \times (4 \times R + 1)$ for the min-approach and $(WG + 2 \times R)^2$ for the max-approach, where *WG* is the work-group size ($WG_x = WG_y = WG$), $R$ is the radius of the block ($R_x = R_y = R$). In Fig. 5, $WG = 8$ and $R = 1$. Thus, the min-approach needs 50 cells, while the max-approach needs 100 cells. We have implemented the *max-approach* in the code generator and we use the *min-approach* as a post-optimization step.

**Step 2** (Loading data into local memory (@*lmLoad*)). When moving data elements from global memory to local memory, multiple passes are needed with `FCTH` [6]. When using the max-approach, we bind one thread to one data element in the central shaded area (outlined by dashed square in Fig. 5(c)). Thereafter, we load the border data into the local space. Nevertheless, loading data with the min-approach is more complicated. Apart from thread masking, we have to deal with *data shuffling* to put the data elements in the right places and thus it is MAP-dependent.

**Step 3** (Accessing local memory (@*lmUse*)). Using data elements in local memory is straightforward. The key is to find the correspondence between the global data index and its local data index. For MAP-407, each thread needs a block of data elements around its thread index (the light-shaded elements in Fig. 5(b) and (c)). Besides, we need to *re-shuffle* the access index when using the min-approach.

By using our code generator, we obtain 66 microbenchmarks (33 using local memory and 33 using global memory only).[4] Our experience shows that the technical difficulty of designing the code generator is dealing with the aforementioned three steps. Given that the use of the min-approach and bank-conflict removal varies from MAP to MAP, we take them as post-step optimizations of the kernels.

## 6. Performance database

### 6.1. Performance metric

We use memory bandwidth as our performance metric. Suppose we have $W \times H$ threads, and each needs $N$ data elements of `type` data type ($N$ is determined by iMAP). We run each kernel and measure the kernel execution time $T$. Then we calculate the bandwidth as $W \times H \times N \times size(type)/T$. We measure the memory bandwidth for cases without ($b$) and with local memory ($B$), use $b$ as the reference, and calculate the memory bandwidth ratio ($mbr = B/b$). If $mbr > 1$, using local memory is beneficial in terms of memory bandwidth; otherwise, using local memory leads to a performance loss.

### 6.2. Experimental setup

We have run and compared the benchmarks on seven platforms, whose configurations are shown in Table 4. When measuring memory bandwidth, we used six data sets ($W \times H$ in Section 4.3): $128 \times 128$, $256 \times 256$, $512 \times 512$, $1024 \times 1024$, $2048 \times 2048$, $4096 \times 4096$. For the Block MAPs, we set the radius to be 3 ($R_x = R_y = 3$) and we expect memory bandwidth to increase with a larger radius. For each measurement, we run 21 iterations (the first iteration as a warm-up run). To avoid data reuse between iterations and cache interference, we flush caches between iterations. Furthermore, we believe that the choice of work-group sizes has an impact on the memory bandwidth. In this work, we set it to be $16 \times 16$.

### 6.3. Performance optimization considerations

#### 6.3.1. The max-/min-approaches

For MAP-407, we compare the *mbr* of the max and min approaches on the seven platforms in Fig. 7. We see that the min-approach is not always performing better than the max-approach. In fact, the min-approach can achieve much better performance on C1060 and X5650, and it performs slightly better on K20m (up to 15%). On C2050, HD7970 and E5-2620, the performance of the min-approach is slightly worse. We also note that the bandwidth suffers around a 40% loss with the min-approach on Xeon Phi. Thus, the overall performance can be significantly influenced by the way of using local memory. When predicting performance, we need to take the design choice into account.

#### 6.3.2. Removing bank-conflicts

As a post-step optimization, we use the *padding* approach to remove bank-conflicts. Taking MAP-204 as an example, we show the performance impacts of removing bank-conflicts in Fig. 8. We see that remov-

[4]The code generator is available: https://github.com/haibo031031/aristotle.

Table 4
Details of the used platforms

| | Platform I | Platform II | Platform III | Platform IV |
|---|---|---|---|---|
| Host | Intel Core i7 920 | Intel Xeon E5620 | Intel Xeon E5620 | Intel Xeon E5620 |
| Host OS | UBUNTU v11.10 | CentOS v6.2 | CentOS v6.2 | CentOS v6.2 |
| Device | NVIDIA Tesla C1060 | NVIDIA Tesla C2050 | NVIDIA Tesla K20m | AMD HD7970 |
| GCC | v4.6.1 | v4.4.6 | v4.4.6 | v4.4.6 |
| OpenCL | CUDA v5.5 | CUDA v5.5 | CUDA v5.5 | AMD APP v2.8 |
| | **Platform V** | **Platform VI** | **Platform VII** | |
| Host | Intel Xeon E5-2620 | Intel Xeon E5-2620 | Intel Xeon X5650 | |
| Host OS | CentOS v6.2 | CentOS v6.2 | CentOS v6.2 | |
| Device | Intel Xeon Phi 5110P | Intel Xeon E5-2620 | Intel Xeon X5650 | |
| GCC | v4.4.6 | v4.4.6 | v4.4.6 | |
| OpenCL | Intel OCL SDK v3.0 | Intel OCL SDK v3.0 | Intel OCL SDK v3.0 | |

ing bank-conflicts on the GPUs (i.e., C1060, C2050, K20m, HD7970) significantly increases the memory bandwidth (up to 7×), while the 'optimization' leads to a performance decrease on the cache-only processors (i.e., Xeon Phi, E5-2620 and X5650). Thus, we conclude that this optimization is specific for processors with a scratch-memory.

### 6.4. Performance database

#### 6.4.1. Database record

After running the microbenchmarks, we obtain a performance database indexed by three items (platform, map, dataset) shown in Fig. 9. Once the index is specified, a query in the database will return a database *record*. Each record consists of the memory bandwidth without local memory ($b$), the memory bandwidth with local memory ($B$), and their ratio (*mbr*).

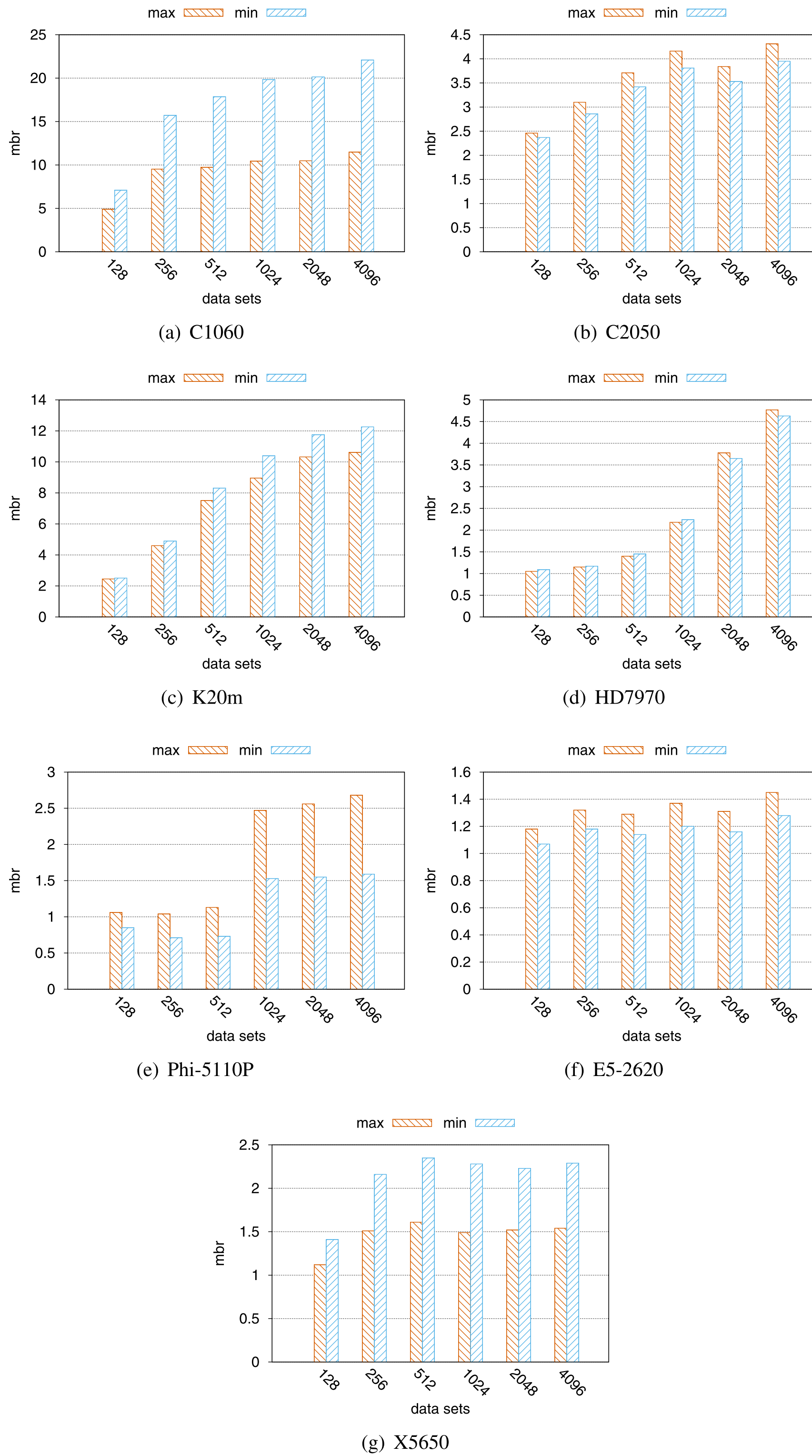
#### 6.4.2. Observations

We run each experiment for 21 times, and calculate the average value and the standard derivation value. For demonstration simplicity, we show the $b$ and $B$ (the average and the standard derivation number) of the performance database (available on-line[5]): the horizontal axis represents the six data sets and the vertical axis represents bandwidth. Overall, we found that the performance benefits of using local memory are heavily dependent on the size of the data sets. In most cases, the bandwidth increases over datasets. Only in a few cases (e.g., the Column MAPs on E5-2620 and X5650), the bandwidth without local memory decreases over datasets. Besides, we make the following observations for each platform:

*C*1060. It differs from other processors in that it has a scratch-pad memory, but no caches. We see that using local memory on C1060 can achieve a memory bandwidth increase for most memory access patterns (29 out of 33 MAPs). When looking into the microbenchmarks, we found that there are two factors leading to the bandwidth increase – data reuse and changes in global memory access orders. As we have mentioned in Section 2, data reuse is a common indicator of using local memory. Taking the Single MAPs for example (see Table 3), we can reuse data for MAPs-(107, 116). Using local memory can also change the memory access order and reduce the number of memory transactions (thus increase the off-chip memory bandwidth). For the Single MAPs, memory access orders of MAPs-(107, 110, 112, 113, 115) are changed when using local memory. For MAP-108, performance is lost ($mbr < 1$) because no data reuse or changes in memory access order appear. The performance loss results from the overhead of using local memory. Although data reuse exists in MAP-109, the data request from the off-chip memory can be serviced in a broadcast manner, and thus using local memory brings no bandwidth improvement. Furthermore, all the Row MAPs, Column MAPs, and Block MAPs can benefit from data reuse, and some of them can even achieve a bandwidth increase due to the common effort of both data reuse and memory access changes.

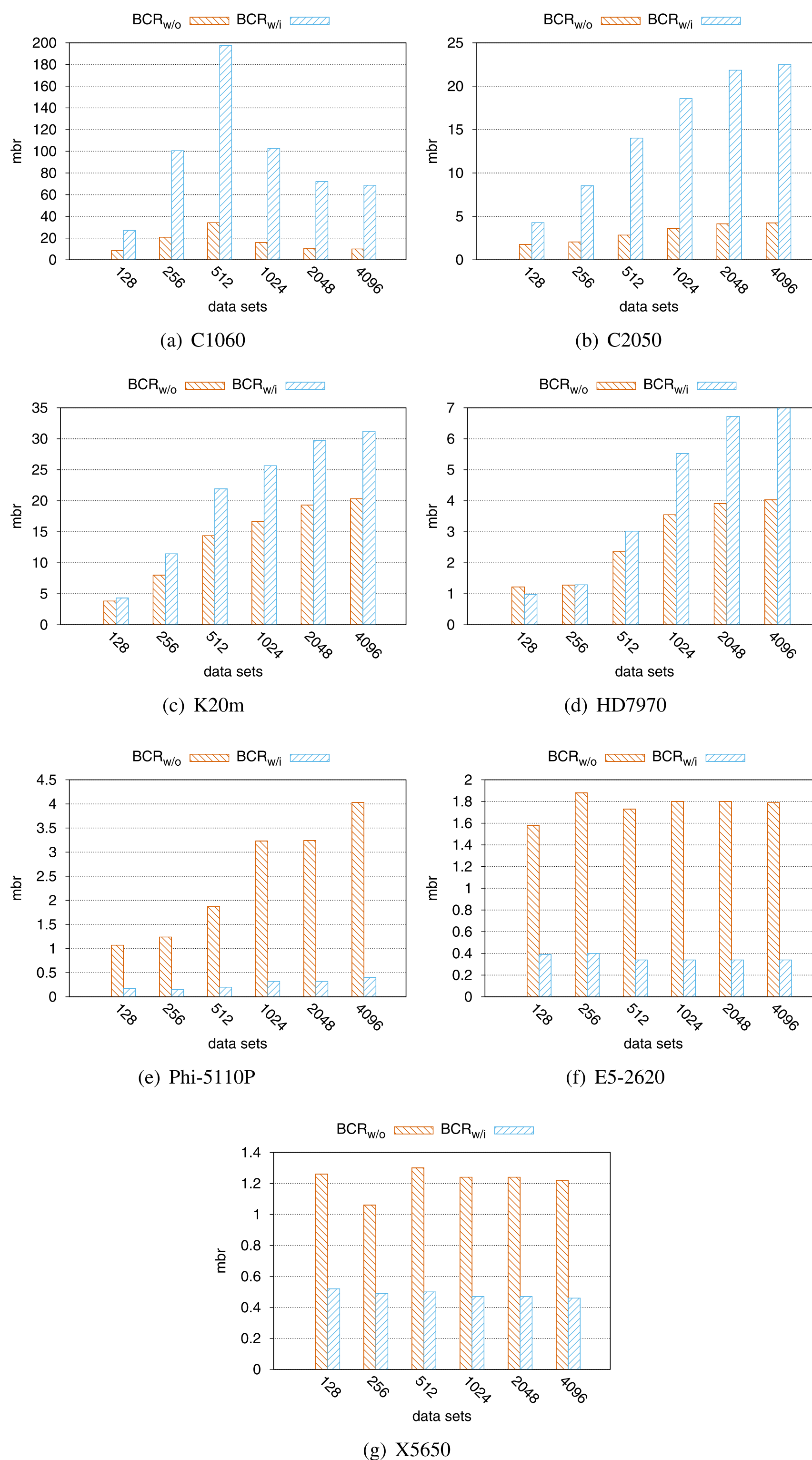
*C*2050 *and K*20*m*. They have not only scratch-pad memories but also caches. Similar to C1060, using local memory on C2050 and K20m is highly beneficial in most cases. On C2050 and K20m, the bandwidths follow the same trends with that on C1060, but the changes are less significant: the older cache-less C1060 benefits much more from local memory than

[5] https://github.com/haibo031031/aristotle/tree/master/pdb.




(a) C1060 (b) C2050 (c) K20m (d) HD7970 (e) Phi-5110P (f) E5-2620 (g) X5650

Fig. 7. Performance comparison of the max/min approaches. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)




(a) C1060 (b) C2050

(c) K20m (d) HD7970

(e) Phi-5110P (f) E5-2620

(g) X5650

Fig. 8. Performance comparison before and after removing bank-conflicts. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)

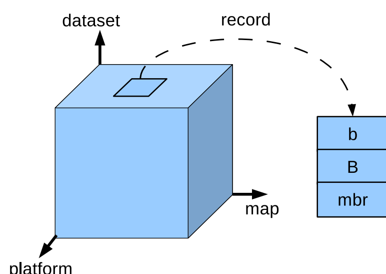



Fig. 9. The database dimensions and its record. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)

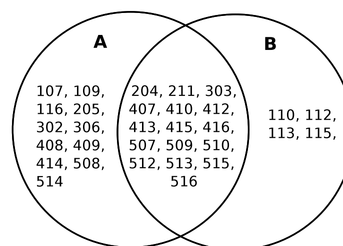



Fig. 10. Performance factors and MAPs distribution.

the newer C2050 and K20m. This happens because the caches will alter the performance benefits of the explicit usage of local memory. In some cases, hardware caches are able to make use of the inherent data locality in the MAP without using scratch-pad memory (see MAPs-116, 508 and 514). In other cases, with more complicated locality patterns, explicit usage of local memory remains beneficial on C2050 and K20m (see MAPs-204, 303, 410, etc.).

*HD*7970. The processor also has both scratch-pad memories and caches. For most MAPs, the performance benefits are less significant and the bandwidth varies a lot over the data sets, which significantly differs from that on NVIDIA GPUs. We believe this is because HD7970 has a different cache architecture and implementation compared to C2050 and K20m.

*Phi*-5110*P*, *E*5-2620 *and X*5650. These processors only have caches on-chip, and implement OpenCL local memory on global memory, in an emulation mode. Thus, using local memory is equivalent to using the off-chip global memory and introduces extra overheads (compared with using global memory directly), which might slow down the execution. However, we get better performance for some MAPs (e.g., Column MAPs) by using local memory (see Table 5). This is due to better caching when using a smaller memory space. We also note that the bandwidth varies more significantly between runs on E5-2620 and X5650 than on Phi-5110P and GPUs (see the error bars in the online figures). Another interesting observation is that using local memory preserves bandwidth on MAPs like MAP-302 while the bandwidth drops over datasets on E5-2620 and X5650. We conclude that data reuse is a must to obtain a bandwidth increase by using local memory on cache-only processors.

*6.4.3. Performance factors analysis*

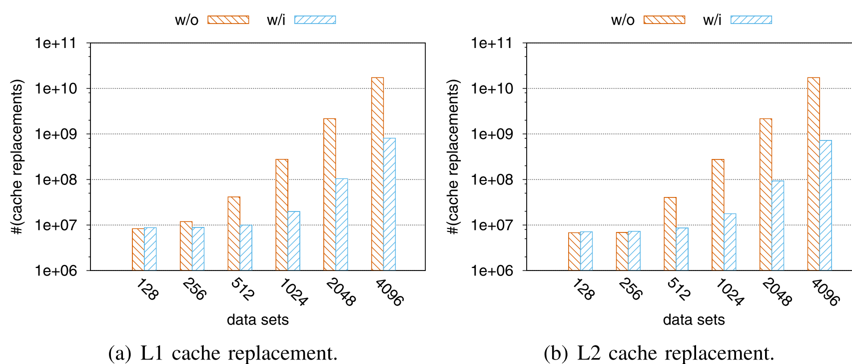
As we have shown, two factors contribute to the memory bandwidth improvement: data reuse (Factor A) and access order changes (Factor B). We analyze the MAPs and identify the factors for each MAP in Fig. 10. We see that 11 MAPs present the potentials of reusing data, while 4 MAPs can benefit from the changes in memory access orders due to the usage of local memory. Typically, data access orders can be changed when loading data from global space to local space. Besides, there are 16 out of the 33 MAPs that can use both of them. Data reuse can be a benefit source for both caches and scratch-pad memories, whereas access order changes does not necessarily lead to a bandwidth increase.

*6.4.4. Architecture-dependent analysis*

We roughly divide the selected processors into three groups: the SPM-only processors, the SPM-Cache processors, and the Cache-only processors. The SPM-only processors (e.g., C1060) have a scratch-pad memory, but have no on-chip caches. Using local memory on such processors can benefit from either data reuse (i.e., less off-chip data movements) or higher effective off-chip bandwidth (shown in Section 2.2.2). For C2050, K20m and HD7970, they have both scratch-pad memories and caches. Using local memory can give a higher bandwidth (than without it) when the MAPs are cache-unfriendly. Otherwise, adopting local memory leads to a performance decease due to the overheads.

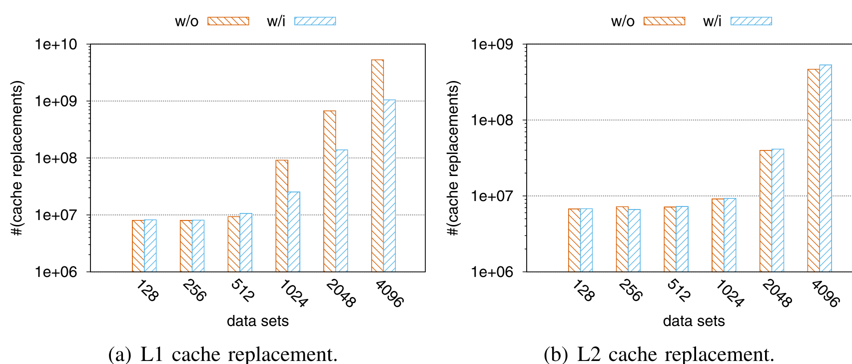
The cache-only processors (Phi-5110P, E5-2620 and X5650) do not have a on-chip scratch-pad memory and local memory is allocated on the global space. However, using local memory can change data layouts (i.e., access orders) and thus can be seen as a ‘software’ optimization technique. By using local memory, the data elements are first loaded into the local space and then accessed within the local space. In this way, we may avoid ‘unnecessary’ cache-line replacements and have a better utilization of caches.

We measure the number of cache-line (L1 and L2) replacements for MAP-302 and MAP-204 on E5-2620 (Figs 11 and 12). For MAP-302, we see that the number of cache replacements is larger without using local




(a) L1 cache replacement. (b) L2 cache replacement.

Fig. 11. The number of cache replacements for MAP-302. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)




(a) L1 cache replacement. (b) L2 cache replacement.

Fig. 12. The number of cache replacements for MAP-204. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140390.)

memory for both L1 cache and L2 cache. Thus, using local memory on MAP-302 is beneficial on E5-2650 especially for large datasets. However, for MAP-204, cache-lines are replaced less frequently when using local memory on the L1 cache while it occurs more often on the L2 cache. We note that using local memory gives a smaller bandwidth for this MAP.

### 6.4.5. *Performance gain/loss distribution*

We define that using local memory has a *similar* performance to that using global memory when $|1.0 - \delta| \leqslant mbr \leqslant |1.0 + \delta|$. Therefore, using local memory has a *gain* performance when $mbr > |1.0 + \delta|$, and using local memory has a *loss* performance when $mbr < |1.0 - \delta|$. We show the overall performance gain/loss distribution in Table 5, where $\delta = 0.05$ (5%). We note that, in most cases on NVIDIA GPUs, using local memory gives us a bandwidth increase. Specifically, the number is around 90% on C1060, and 80% on C2050 and K20m. Thus, using caches partially 'cancels' the benefits of using local memory. On AMD HD7970, over half of the MAPs have a similar performance between with and without adopting local memory for the small datasets, while the number becomes smaller for the larger datasets and more MAPs can benefit from using local memory. On the cache-only processors (Phi-5110P, E5-2620, X5650), using local memory leads to a performance decrease for around half of the MAPs for small datasets. For large datasets, we note less MAPs on Phi-5110P but more MAPs on E5-2620 and X5650 that has a performance decrease by using local memory. This is mainly due to the fact that they have a difference cache architectures (i.e., Xeon Phi has a distributed last-level cache while the other processors have a unified one). We recommend using local memory for the *better* MAPs based on the guidelines mentioned in Section 5; for the cases with little or no bandwidth increase (or even bandwidth decrease), using local memory is not recommended due to the low ratio between performance gain and programming effort.

Table 5
Performance gain/loss distribution ($\delta = 0.05$)

| | | C1060 | C2050 | K20m | HD7970 | Phi-5110P | E5-2620 | X5650 |
|---|---|---|---|---|---|---|---|---|
| 128 | Gain | 27 (81%) | 21 (63%) | 20 (60%) | 6 (18%) | 7 (21%) | 12 (36%) | 9 (27%) |
| | Loss | 1 (3%) | 4 (12%) | 0 (%) | 5 (15%) | 16 (48%) | 17 (51%) | 17 (51%) |
| | Similar | 5 (15%) | 8 (24%) | 13 (39%) | 22 (66%) | 10 (30%) | 4 (12%) | 7 (21%) |
| 256 | Gain | 29 (87%) | 25 (75%) | 26 (78%) | 13 (39%) | 7 (21%) | 9 (27%) | 9 (27%) |
| | Loss | 2 (6%) | 5 (15%) | 4 (12%) | 3 (9%) | 20 (60%) | 22 (66%) | 20 (60%) |
| | Similar | 2 (6%) | 3 (9%) | 3 (9%) | 17 (51%) | 6 (18%) | 2 (6%) | 4 (12%) |
| 512 | Gain | 29 (87%) | 26 (78%) | 26 (78%) | 13 (39%) | 8 (24%) | 7 (21%) | 11 (33%) |
| | Loss | 3 (9%) | 6 (18%) | 6 (18%) | 3 (9%) | 18 (54%) | 20 (60%) | 21 (63%) |
| | Similar | 1 (3%) | 1 (3%) | 1 (3%) | 17 (51%) | 7 (21%) | 6 (18%) | 1 (3%) |
| 1024 | Gain | 29 (87%) | 26 (78%) | 26 (78%) | 14 (42%) | 16 (48%) | 8 (24%) | 11 (33%) |
| | Loss | 4 (12%) | 6 (18%) | 6 (18%) | 1 (3%) | 14 (42%) | 24 (72%) | 19 (57%) |
| | Similar | 0 (%) | 1 (3%) | 1 (3%) | 18 (54%) | 3 (9%) | 1 (3%) | 3 (9%) |
| 2048 | Gain | 29 (87%) | 26 (78%) | 26 (78%) | 15 (45%) | 17 (51%) | 8 (24%) | 8 (24%) |
| | Loss | 4 (12%) | 6 (18%) | 6 (18%) | 5 (15%) | 15 (45%) | 24 (72%) | 21 (63%) |
| | Similar | 0 (%) | 1 (3%) | 1 (3%) | 13 (39%) | 1 (3%) | 1 (3%) | 4 (12%) |
| 4096 | Gain | 29 (87%) | 26 (78%) | 26 (78%) | 16 (48%) | 16 (48%) | 9 (27%) | 12 (36%) |
| | Loss | 4 (12%) | 6 (18%) | 6 (18%) | 5 (15%) | 15 (45%) | 22 (66%) | 20 (60%) |
| | Similar | 0 (%) | 1 (3%) | 1 (3%) | 12 (36%) | 2 (6%) | 2 (6%) | 1 (3%) |

## 7. Composing MAP impacts

We have quantified the performance impacts of using local memory for isolated MAPs and a simple query in the database can tell us the performance benefits. However, a real-world kernel often has multiple data structures (and memory access patterns). In this section, we propose composing rules in the presence of multiple MAPs to give the performing order of using local memory.

For a given *MAP*, let $\oplus$ represent that using local memory brings a 'positive' performance impact (positive MAP) and let $\ominus$ represent that using local memory gives a 'negative' performance impact (negative MAP). Assume we have two MAPs (two data structures in a kernel): *MAP1* and *MAP2*, which can be the same pattern or two different patterns. For each MAP, we have two choices: $l$ – choose to use local memory, and $g$ – choose not to use local memory (thus using global memory). Then we can obtain four versions of code for this kernel: $(g, g)$, $(g, l)$, $(l, g)$ and $(l, l)$. We need to pick the most efficient choice among the four.

We use two metrics to evaluate the efficiency: *performance* and *programming efforts*, of which *performance* is taken as our first priority. In this work, we consider an effort of enabling local memory usage for a data structure as the unit of programming effort. Ideally, we prefer an efficient solution with less programming effort. To compose MAP impacts, we propose and analyze the following rules.

**Rule 5.1.** $\ominus + \ominus \to (g, g)$.

*Analysis*. For either *MAP1* or *MAP2*, using local memory leads to a performance loss. When composing them, we cannot find any sources of a performance gain. Thus, we choose not to use local memory for both of them.

**Rule 5.2.** $\oplus + \ominus \to (l, g)$, $\ominus + \oplus \to (g, l)$.

*Analysis*. Suppose *MAP1* can benefit from using local memory ($\oplus$), while *MAP2* suffers a performance loss ($\ominus$). Since using local memory on *MAP2* brings us no performance gain, we choose not to use local memory on it. Next, let us consider *MAP1*. Further suppose that we need $D_1$ data elements for *MAP1* and $D_2$ data elements for *MAP2*, and their bandwidths are $W_1$ and $W_2$, respectively. Let $W$ be the overall bandwidth and $T$ represent the data transfer time. Thus, we can obtain

$$T = \frac{D_1 + D_2}{W(W_1, W_2)}.$$

As we have analyzed, the performance gain of using local memory comes from two factors: either the

decrease of data amount ($D$), and/or the increase of global memory bandwidth ($W$). Thus, we need to consider two cases.

**Case 1.** Using local memory on *MAP1* decreases $D_1$. In such a case, we need to move less data from global memory ($D$), thus posing less contention for the shared resources (e.g., channels and ports) and leaving more chances for *MAP2* to transfer data. Therefore, using local memory on *MAP1* will improve the overall performance, i.e., $\phi(l, g) > \phi(g, g)$.

**Case 2.** Using local memory on *MAP1* increases $W_1$. In case of a under-utilized bandwidth, we can better utilize shared resources (ports and channels) and thus have a better performance. Once we reach the maximum achievable bandwidth, using local memory further brings us no performance gain. In other words, using local memory on *MAP1* can guarantee that $\phi(l, g) \geqslant \phi(g, g)$.

Therefore, we choose $(l, g)$ in this case. Likewise, we choose $(g, l)$ for the $\ominus + \oplus$ combination.

**Rule 5.3.** $\oplus + \oplus \to (?, l)$.

*Analysis.* As we infer from *Rule II*, we can guarantee that $\phi(l, g) \geqslant \phi(g, g)$. Thereafter, we need to take a decision of enabling local memory on *MAP2*. We can see similar performance benefits as shown in *Rule II*. However, using local memory on *MAP2* also increases the amount of used local memory and thus may reach the maximum limit on the device. Therefore, we need to check whether there remains enough local space beforehand. If there is enough local space for MAP2, we will choose to perform allocation for it.

**Rule 5.4.** $\underbrace{\oplus + \oplus + \cdots + \oplus}_{m} + \underbrace{\ominus + \ominus + \cdots + \ominus}_{n} \to (\underbrace{?, ?, \ldots, l}_{m}, \underbrace{g, g, \ldots, g}_{n})$.

*Analysis.* Assume that, when using local memory, we divide the *MAPs* into two groups based on the performance benefits: $m$ *MAPs* can benefit from using local memory while $n$ *MAPs* suffers in performance. By iteratively using Rule 5.1, we choose not to use local memory on these $n$ negative *MAPs*. According to Rule 5.2, we use local memory on the right-most *MAP*. Thereafter, it is unclear whether to use local memory or not based on Rule 5.3. Thus, this rule is a derivation of Rules 5.1–5.3.

Up to now, there remains one question: *which positive MAP do we select first?* Different MAPs may differ in performance benefits due to the MAP feature and its run-time dataset. Suppose when using local memory on MAP1, the performance benefit is $mbr_1$ and the dataset is $D_1$; when using local memory on MAP2, the performance benefits is $mbr_2$ and the dataset is $D_2$. Then we can calculate the *performing order weight* $\omega$. When using local memory, we will select the MAP with the largest $\omega$ until we do not have enough local space

$$\omega_1 = \frac{D_1}{D_1 + D_2} \times mbr1,$$
$$\omega_2 = \frac{D_2}{D_1 + D_2} \times mbr2.$$

## 8. Composing rules validation

### 8.1. A MAP composer

To validate our composing rules, we compose MAPs based on the code generator mentioned in Section 5. When multiple MAPs are used in a kernel, we consider the use of local memory in an incremental manner. In other words, the kernel template takes multiple MAPs as input and we use local memory on them one by one. For 2 MAPs, we have built a composer and show its structure in Fig. 13.[6] The composer generates three code versions: (v0) without using local memory, (v1) a code version of using local memory on MAP1, and (v2) a code version of using local memory on both MAP1 and MAP2. Thus, when we have $N$ MAPs (they

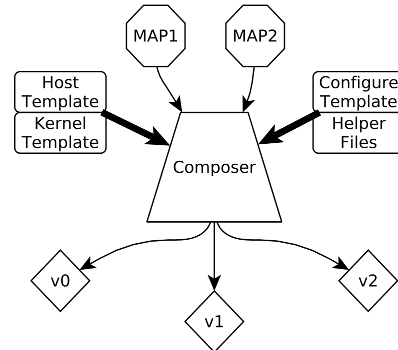



Fig. 13. The composer architecture.

[6]The composer is available: https://github.com/haibo031031/aristotle.

can be the same or different), the composer will generate $N + 1$ versions of code.

We validate our proposed rules for each use of local memory. We take the test case as a successful prediction when the results meet with those from the composing rules. We calculate the prediction accuracy as the number of successful tests divided by the total number of tests. We assume that a test fails when we observe one of the follows:

(1) A positive MAP gives a performance degradation from using local memory.
(2) A negative MAP gives a performance increase from using local memory.

### *8.2. Rule validation*

We use 7 platforms and 6 datasets (see Section 6.2), and use 2 data structures which are of the same MAP or 2 different MAPs.[7] Note that we can guarantee there is sufficient local space when considering 2 data structures. With 33 MAPs, we need to evaluate 2178 ($33 \times 33 \times 2$) test cases for each platform and dataset. Thus, we can evaluate the performing order $\omega$ in an exhaustive manner.

Our validation results are shown in Table 6. We see that the rules holds with an accuracy of around 90% on NVIDIA GPUs, while the number ranges from 55% to 75% on the AMD GPU. On the cache-only processors (Xeon Phi, E5-2620), the prediction accuracy is up to 80%, while we note that the rules see a relatively low accuracy on X5650. We believe this is because of the cache interferences between the data structures.

### *8.3. Using Aristotle*

Given a kernel, users first need to abstract the MAP for each data structure. Depending on the given platform and the MAP, we query the performance database. If it is beneficial, we will perform code transformation to enable local memory on the input kernel. Otherwise, we keep the original kernel code. When the given kernel has multiple MAPs, we need to calculate the performing order weight $\omega$ based on the isolated *mbr*. The composing rules (in Section 7) provide users with a reference on how to compose multiple MAPs. For NVIDIA GPUs, the prediction accuracy is high, but it is relatively low on the cache-only processors such as X5650. We recommend the use of local memory (as a software optimization technique) for the MAPs that have a large bandwidth benefit (e.g., $mbr > 1.5$). For the MAPs that show smaller bandwidth benefits, using local memory is not recommended.

Table 6

Rule validation results (%)

| | 128 | 256 | 512 | 1024 | 2048 | 4096 |
|---|---|---|---|---|---|---|
| C1060 | 90 | 95 | 94 | 93 | 92 | 93 |
| C2050 | 93 | 94 | 94 | 93 | 92 | 91 |
| K20m | 90 | 91 | 91 | 90 | 89 | 88 |
| HD7970 | 55 | 63 | 69 | 69 | 75 | 66 |
| Phi-5110P | 74 | 80 | 81 | 77 | 79 | 83 |
| E5-2620 | 61 | 73 | 78 | 80 | 81 | 80 |
| X5650 | 65 | 68 | 72 | 70 | 73 | 65 |

[7] Running the validation experiments for 2 MAPs takes 2–6 days per platform and we cannot afford to validate more MAP composition.

## 9. Related work

In this section, we discuss prior work on benefits prediction and code transformation of using local memory.

Enabling local memory has been studied extensively for the SPMs on GPUs. In [14], Kandemir et al. propose a scratch-pad memory design and optimization framework. In this framework, the compiler has a central role in the sense that it manages the flow of data across a given hierarchy (by staging computation and data). In [27], Sumesh Udayakumaran et al. propose a highly predictable, low overhead, and dynamic memory-allocation strategy for embedded systems with scratch pad memory. The scheme can follow changing working sets by moving data from scratchpad to DRAM under compiler control. In [8], the authors present a method for computing precisely which memory cells are reused due to temporal locality of scientific codes. By way of precise data dependence information, they can determine exactly when to copy a value to fast memory, when to copy an updated value back to main memory and when to relocate a value in fast memory. In [29], the authors present a source-to-source compiler called PPCG, which accelerates computations from any static control loop nest of a sequential program and generates multiple CUDA kernels. In particular, they take care of the use of on-chip shared memory and consider two criterion: data reuse and coalescing. In [28], Nicolas Vasilache et al. propose a set of automated techniques to optimize memory reuse in programs with explicitly managed memory.

In addition, Pouchet et al. 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 [25]. Therefore, most studies focus on identifying data reuse (e.g., using a polyhedral model) when enabling local memory. We predict the benefits of using local memory in a generic manner, which is necessary when we have diverse computing devices. Further, our performance indicator can serve as the input of the aforementioned code transformers.

Alternatively, several API-based approaches have also been proposed to enable local memory. In [4], the authors present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memories of GPUs. In [6], we present a user-friendly API, ELMO, based on identifying patterns of local memory usage.

A more generic related topic is auto-tuning. Generally, there are two types of auto-tuning: empirical optimization [7,18,20,21] and model-driven optimization [2,10,16,30]. Although empirical optimization techniques giving the optimal performance, it generates a large number of parameterized code variants and the time cost of searching for the best code variant makes it less attractive. In contrast, model-driven optimizations self-tune implementation-related parameters to obtain optimal performance. Using model-driven auto-tuning typically has an O(1) cost, since the parameters can be derived from the analytical model. However, it may not give optimal performance, because analytical models are only simplified abstractions of architectures and/or applications. Our approach relates both: we use modelling to build our database, and use the database to potentially prune the search space of empirical autotuners.

## 10. Conclusions and future work

Architecture diversity and application implementation differences make the performance benefits of using local memory much less predictable than expected. In this work, we presented a benchmark-based approach (*Aristotle*) to tackle this issue starting with the memory access patterns (MAPs). For each such MAP, we generated benchmarks for a naive version (without local memory) and an optimized one (using local memory). We evaluated the microbenchmarks on the NVIDIA GPUs (C1060, C2050, K20m), AMD GPUs (HD7970), Intel Xeon CPUs (E5-2620 and X5650), and Intel Xeon Phi 5110P, and obtained a performance database.

By analyzing the memory access patterns and the performance impacts of using local memory, we have found that both *data reuse* and *changes in access order* may contribute to the effective bandwidth increase. On the processors with both scratch-pad memories and caches, the performance benefits of using local memory in OpenCL kernels are less significant. Furthermore, using local memory on the cache-only processors (e.g., the traditional multicore CPUs) can be seen as a software optimization and might be efficient by better utilizing caches.

This is the first extensive, systematic study of local memory impacts based on generalized MAPs. Not only can this work provide essential information for performance prediction with database queries, but it can also give a performance indicator of local memory usage. Further, we presented four rules to generate the performing order of using local memory when we have multiple data structures. Our results validated the usefulness of our composing rules on GPU architectures in particular. Meanwhile the prediction accuracy is relatively low on the cache-only processors largely due to the cache interference between multiple data structures. We believe that this issue is impossible (or difficult at least) to fix because of the dynamic nature of cache interference.

We note that for new emerging platforms (with OpenCL support), the database can be easily extended: one can simply use the microbenchmarking and logging tools to expand it. The performance database, together with the composing rules will give us an indicator of whether or not to use local memory.

For future work, we want to implement an autotuner based on the performance database and the derived conclusions. This auto-tuner will enable or reverse the use of local memory based on the performance (memory bandwidth) benefits. Furthermore, the configuration of caches and SPMs needs further exploration from the perspective of computer architects – which configurations are suitable (considering both power consumption and performance) for which specific class of applications?

## Acknowledgements

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] AMD Inc., *AMD Accelerated Parallel Processing – OpenCL*, May 2012.

[2] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp and W. Mei, An adaptive performance modeling tool for GPU architectures, in: *PPoPP'10*, ACM, New York, NY, USA, 2010, pp. 105–114.

[3] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev and P. Sadayappan, Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories, in: *Proceedings of PPoPP*, 2008.

[4] 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*, ACM, New York, NY, USA, 2011.

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

[6] 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.

[7] D. Grewe and A. Lokhmotov, Automatically generating and tuning GPU code for sparse matrix–vector multiplication from a high-level representation, in: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU2011)*, GPGPU-4, ACM, New York, NY, USA, March 2011.

[8] A. Grösslinger, Precise management of scratchpad memories for localising array accesses in scientific codes, in: *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC'09*, Vol. 5501, Springer-Verlag, Berlin/Heidelberg, 2009, pp. 236–250.

[9] 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 the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT'10*, ACM, New York, NY, USA, 2010, pp. 205–216.

[10] S. Hong and H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: *ISCA'09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, 2009, pp. 152–163.

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

[12] I. Issenin, E. Brockmeyer, M. Miranda and N. Dutt, DRDU: A data reuse analysis technique for efficient scratch-pad memory management, *ACM Trans. Des. Autom. Electron. Syst.* **12** (2007), 15:1–28.

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

[14] M. Kandemir and A. Choudhary, Compiler-directed scratch pad memory hierarchy design and management, in: *Proceedings of DAC*, ACM, 2002.

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

[16] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan and K. Srinathan, A performance prediction model for the CUDA GPGPU platform, in: *Proceedings of 2009 International Conference on High Performance Computing (HiPC)*, December 2009, pp. 463–472.

[17] S.-t. Leung and J. Zahorjan, Optimizing data locality by array restructuring, Tech. Rep. TR 95-09-01, University of Washington, 1995.

[18] Y. Li, J. Dongarra and S. Tomov, A note on auto-tuning GEMM for GPUs, in: *Proceedings of the 9th International Conference on Computational Science: Part I, ICCS'09*, Lecture Notes in Computer Science, Vol. 5544, Springer, Berlin/Heidelberg, 2009, pp. 884–892.

[19] A. Marongiu and L. Benini, An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs, *IEEE Trans. Comput.* **61** (2012), 222–236.

[20] A. Monakov, A. Lokhmotov and A. Avetisyan, Automatically tuning sparse Matrix–Vector multiplication for GPU architectures, in: *Proceedings of the 5th International Conferences on High Performance Embedded Architectures and Compilers (HiPEAC 2010)*, Lecture Notes in Computer Science, Vol. 5952, Springer, Berlin/Heidelberg, 2010, pp. 111–125.

[21] A. Nukada and S. Matsuoka, Auto-tuning 3-D FFT library for CUDA GPUs, in: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09*, ACM, November 2009.

[22] NVIDIA Inc., *NVIDIA CUDA C Programming Guide Version 4.1*, 2011.

[23] NVIDIA Inc., CUDA Toolkit 4.1, February 2012, available at: http://developer.nvidia.com/cuda-toolkit-41.

[24] S.A. Ostadzadeh, R.J. Meeuws, C. Galuzzi and K. Bertels, QUAD: a memory access pattern analyser, in: *Proceedings of the 6th International Conference on Reconfigurable Computing: Architectures, Tools and Applications, ARC'10*, Berlin/Heidelberg, 2010, pp. 269–281.

[25] 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*, ACM, New York, NY, USA, 2013, pp. 29–38.

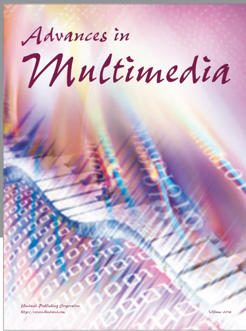
[26] S. Saidi, P. Tendulkar, T. Lepley and O. Maler, Optimizing explicit data transfers for data parallel applications on the cell architecture, *ACM Trans. Archit. Code Optim.* **8** (2012), 37:1–20.

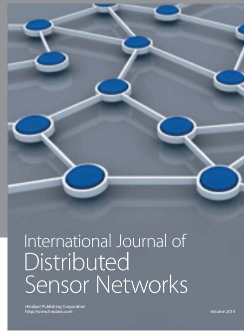
[27] S. Udayakumaran, A. Dominguez and R. Barua, Dynamic allocation for scratch-pad memory using compile-time decisions, *ACM Trans. Embed. Comput. Syst.* **5** (2006), 472–511.

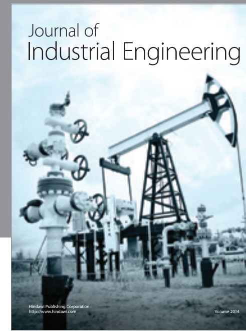
[28] N. Vasilache, M. Baskaran, B. Meister and R. Lethin, Memory reuse optimizations in the R-Stream compiler, in: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6*, ACM, New York, NY, USA, 2013, pp. 42–53.

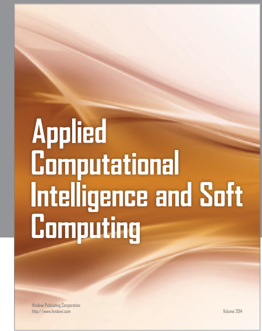
[29] S. Verdoolaege, J.C. Juega, A. Cohen, J.I. Gómez, C. Tenllado and F. Catthoor, Polyhedral parallel code generation for CUDA, *ACM Trans. Archit. Code Optim.* **9** (2013), 54:1–23.

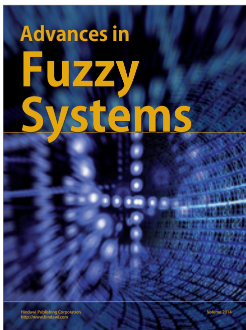
[30] Y. Zhang and J.D. Owens, A quantitative performance analysis model for GPU architectures, in: *HPCA 2011*, February 2011.











Hindawi

Submit your manuscripts at
http://www.hindawi.com