Quantifying the Performance Impacts of Using Local Memory for Many-Core Processors

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

Ana Lucia Varbanescu
University of Amsterdam, The Netherlands
Email: 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 further 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. To show how our database can be used to provide information on local memory usage, we demonstrate a scenario for performance prediction: given an application, its MAPs, and a platform, a query in the database can indicate the performance impact of using local memory for the given application. 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.

Index Terms—Micro-Benchmarking, Local Memory, Memory Access Pattern, OpenCL, Performance Prediction.

I. INTRODUCTION

To alleviate the memory bottleneck, modern many-core processors use programmer-managed scratch-pad memory. OpenCL [1], the standard proposed by the Kronous Group for programming many-cores, recognizes this under the name of local memory 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 (e.g., NVIDIA maps local memory onto shared memory, and Intel maps it to global memory in emulation mode). Because local memory is often situated on-chip, it is much faster than the global memory. Thus, proper use of local memory often leads to higher memory bandwidth and performance improvement.

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

Typically, programmers use the trial-and-error approach to enable local memory: taking a naive kernel, they translate the code into an “optimized” version that uses local memory and then evaluate its effectiveness. This is a time-consuming process, as programmers have to address, in their OpenCL code, challenges like (1) geometry mismatch, (2) work-items 2 masking and binding switches, and (3) inefficient local memory organization [2]. We argue that solving these problems requires a lot of effort to be spent on non-computational and non-functional details, which hinders productivity. Therefore, it is desirable to determine the performance gain/loss of using local memory before enabling it. This will save a lot of programming effort.

The impact of local memory usage on performance is not easy to predict. For example, data reuse is a commonly recognized source of performance gains of using local memory [3] [4] [5] [6]. However, we found that data reuse does not always make local memory effective (see Section II-A1). On the other hand, even without data reuse, we can still achieve bandwidth increase by using local memory (see Section II-A2). Furthermore, in the case of CPUs, the off-chip placement of the local memory makes programmers choose not to use it [7]. Nevertheless, we have shown that properly using local memory on CPUs can lead to performance improvement (see Section II-A3). These examples prove that the performance benefits of using local memory on various architectures and applications are much less predictable than expected.

In this paper, we address the issue of performance unpredictability when using local memory in a two-stage approach: quantification and composition. Specifically, we develop a benchmark-based approach to quantify the performance impacts of using local memory for 34 different 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. Thereafter, we can obtain the performance benefits of using local memory in practice by querying the database. We further need to compose these results to estimate the benefits for more complex applications. For example, assume the user needs to improve the performance of a convolution filter in image processing. Once the

2In the context, ‘work-item’ is synonymous with ‘thread’, and we use them interchangeably.
application is identified as having MAP-408, the query in the database will show that local memory is useful for this kernel on NVIDIA GTX580, and a performance improvement of 83% can be expected for a 7 × 7 convolution kernel (see Figure 8). Similarly, for a matrix multiplication (C = A × B), the kernel requires two input matrices (A and B) with two different MAPs. Thus, a benefit composition of the benefits achieved by the two input matrices is needed (the specific approach is illustrated in Section VI). Our experimental data shows that the impact quantification and composition approach can deliver promising results.

To the best of our knowledge, this is the first extensive and systematic study of local memory impact starting from MAPs. Consequently, we believe our main contributions are (1) the performance quantification, (2) the microbenchmarking technique, and (3) the query-based performance prediction. For now, our work is targeted at OpenCL-compliant processors, but our empirical approach can also be used for other processors with a scratch-pad memory.

The paper is organized as follows: Our approach and the benchmarking framework are presented in Section II. We extend a mathematical model to describe memory access patterns and derive a set of MAPs in Section III. We explore the design space of using local memory and generate benchmarks using a code template in Section IV. We generate a performance database by running the microbenchmarks on four typically used platforms in Section V. In Section VI, we discuss how to predict performance gain or loss based on our performance database. We present related work in Section VII and we summarize our findings in Section VIII.

II. MOTIVATION AND APPROACH

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

A. Three Observations

1) Data Reuse ≠ 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 I shows the memory bandwidth when running NBODY [8] 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.

<table>
<thead>
<tr>
<th>Datasize</th>
<th>64x64</th>
<th>128x128</th>
<th>256x256</th>
<th>512x512</th>
<th>1024x1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM_W/o</td>
<td>617.50</td>
<td>638.43</td>
<td>640.00</td>
<td>614.42</td>
<td>589.95</td>
</tr>
<tr>
<td>LM_W/i</td>
<td>512.44</td>
<td>495.28</td>
<td>378.04</td>
<td>318.54</td>
<td>220.64</td>
</tr>
<tr>
<td>Loss (%)</td>
<td>16.47</td>
<td>22.18</td>
<td>20.11</td>
<td>15.87</td>
<td>11.75</td>
</tr>
</tbody>
</table>

2) No Data Reuse ≠ Performance Loss: Let us consider data movements among registers, local memory, and global memory. Suppose we have N compute units and the bandwidth of local memory access is Wl. An application requires D data elements to be moved when using global memory only (with a bandwidth of Wg), and D’ data elements to be moved from global memory to local memory (with a bandwidth of Wl). We compute the time of data movement with (Tw/o) and without local memory (Tw/i) as shown in Figure 1. 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 (Wg > Wl). 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. With local memory, we can ensure coalescing for both input and output memory access, and thus improve the bandwidth.

3) Local Memory Use on CPUs ≠ Performance Loss: At the moment of writing, local memory is allocated on the main memory space on CPUs. Thus, it is not recommended to use local memory on CPUs [7]. However, we have found that this does not always hold. Table II shows the memory bandwidth of a convolution kernel on Intel E5620 (a 6-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.

<table>
<thead>
<tr>
<th>Datasize</th>
<th>64x64</th>
<th>128x128</th>
<th>256x256</th>
<th>512x512</th>
<th>1024x1024</th>
<th>2048x2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM_W/o</td>
<td>6.11</td>
<td>7.77</td>
<td>7.81</td>
<td>8.06</td>
<td>8.13</td>
<td>8.15</td>
</tr>
</tbody>
</table>
Based on all these observations (Section II-A1 to Section II-A3), 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 modeling to generate microbenchmarks, and use traditional performance measurement to quantify local memory usefulness.

B. Our Approach

Figure 2 shows our microbenchmarking framework. Given a memory access pattern, we generate benchmark kernels without and with local memory. Then we evaluate the benchmarks on typically used many-core processors and generate a performance database.

![Benchmarking Framework](image)

Fig. 2. Benchmarking Framework.

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 application performance. Furthermore, the memory access patterns can be manually identified from input kernels [9], or automatically abstracted during runtime [10], and are, for now, outside of the scope of this work. Instead, we focus on the description (Section III), the development (Section IV), and evaluation (Section V) of the MAP-based microbenchmarks.

III. MAP Description

We express a MAP as a memory access sequence, which allows us to represent discrete memory references and loops. Our approach is based on the notation in [9], [11]. 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 in 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.

A. The notation

According to [9], 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 Equation 1. 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.

\[
\vec{s} = \vec{M}\vec{i} + \vec{o},
\]  

We have adapted this notation to express our specific range of MAPs - see Equation 2.

\[
\vec{s} = eMAP + iMAP = M\vec{t}\vec{i} + iMAP,
\]  

In this new notation, we have clearly separated the inter-thread \((eMAP)\) and intra-thread \((iMAP)\) components. We are able to do so by focusing on 2D thread organization: \(M\) becomes a mapping \(2 \times 2\) matrix of the data/processing to the threads. The \(iMAP\) component is a vector representation of the intra-thread access pattern. Intuitively, \(eMAP\) generates a base access index for each thread, while \(iMAP\) provides an offset which represents the distance from the base address.

We further rewrite Equation 2 to Equation 3, and we use this form to exhaustively generate our benchmarks.

\[
\vec{s} = \begin{bmatrix} M_{00} & M_{01} & M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \\ t_y \\ t_x \end{bmatrix} + \begin{bmatrix} iMAP_0 \\ iMAP_1 \end{bmatrix}
\]  

B. eMAP

When \(M_{00}, M_{01}, M_{10}, M_{11} \in \{0, 1\}\), we generate 16 cases of eMAP (shown in Figure 3).

![eMAP cases](image)

Fig. 3. eMAP cases (numbered 01 to 16 in the left-right and top-down order). E.g., the shaded MAP is #14.

As we have mentioned, eMAP encodes the base index of the memory references for each thread (note that in this context, the iMAP will in fact be used to calculate the memory references offset(s) for each thread). For example, Figure 4 shows the base index of eMAP-14 (the one shaded in Figure 3) 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 \(iMAP = \begin{bmatrix} 0 \\ 0 \end{bmatrix}\). In this case, consecutive work-items in the x dimension will access continuous 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 (Figure 4(a)).
When \( M_{00}, M_{01}, M_{10}, M_{11} \not\in \{0, 1\} \), the eMAPs become more complex. When \( M_{00} = 2 \), we see (Figure 4(b)) ‘gaps’ between rows due to the larger stride, compared with eMAP14. We can imagine that any non-unit stride will introduce such ‘gaps’. For now, our work only considers \([0,1]\) cases (Figure 3). We believe the extension to larger strides will not bring additional confusion to the methodology. However, it will lead to cases very rarely seen in real applications and a large increase in the experimental time. This step is left for future work, once our methodology is thoroughly validated.

### C. 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 [2] - namely, Single, Row, Column, Block, and Neighbor - and be briefly describe them:

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

The iMAP representations are listed as follows:

\[
\text{Single : } \ iMAP = \{ [0] \}
\]

\[
\text{Row : } \ iMAP = \{ [o] \ | \ 0 \leq i < W, i \in N \}
\]

\[
\text{Column : } \ iMAP = \{ [j] \ | \ 0 \leq j < H, j \in N \}
\]

\[
\text{Block : } \ iMAP = \{ [j] \mid -R_x \leq i \leq R_x, i \in N; -R_y \leq j \leq R_y, j \in N \}
\]

\[
\text{Neighbor : } \ iMAP = \{ [-1], [0], [0], [1] \}
\]

### D. MAP = eMAP+iMAP

Once eMAP and iMAP are specified, we should get 80 \((16 \times 5)\) memory access patterns (MAPs), and thus need to generate and evaluate 80 microbenchmarks. The name of each MAP is a concatenation of the iMAP and eMAP numbers. For example, the iMAP of MAP-407 is Block (4) and its eMAP index is 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 underspecified (resulting in non-interesting cases) or overspecified (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:

- **(1) Single**: eMAP has to specify a dependency on both \( t_x \) and \( t_y \). when \( \text{eMAP} = \{ [0] \} \) and \( \text{iMAP} = \{ [0] \} \), the MAP represents an impossible map - either under or overspecified.

- **(2) Row**: when \( \text{eMAP} = \{ [0] \} \) and \( \text{iMAP} = \{ [0] \} \), the MAP represents an impossible map - either under or overspecified.

- **(3) Column**: when \( \text{eMAP} = \{ [0] \} \) and \( \text{iMAP} = \{ [0] \} \), the MAP represents an impossible map - either under or overspecified.

- **(4) Block**: when \( \text{eMAP} = \{ [0] \} \) and \( \text{iMAP} = \{ [0] \} \), the MAP represents an impossible map - either under or overspecified.

- **(5) Neighbor**: when \( \text{eMAP} = \{ [-1], [0], [0], [1] \} \) and \( \text{iMAP} = \{ [-1], [0], [0], [1] \} \), the MAP represents an impossible map - either under or overspecified.

### Table III

<table>
<thead>
<tr>
<th>Single</th>
<th>Row</th>
<th>Column</th>
<th>Block</th>
<th>Neighbor</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>401</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td></td>
<td></td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td></td>
<td></td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>204</td>
<td></td>
<td>205</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>205</td>
<td></td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>206</td>
<td></td>
<td>207</td>
<td>507</td>
</tr>
<tr>
<td>07</td>
<td>107</td>
<td></td>
<td>407</td>
<td>507</td>
</tr>
<tr>
<td>08</td>
<td>108</td>
<td></td>
<td>408</td>
<td>508</td>
</tr>
<tr>
<td>09</td>
<td>109</td>
<td></td>
<td>409</td>
<td>509</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td></td>
<td>410</td>
<td>510</td>
</tr>
<tr>
<td>11</td>
<td>211</td>
<td></td>
<td>411</td>
<td>512</td>
</tr>
<tr>
<td>12</td>
<td>112</td>
<td></td>
<td>412</td>
<td>512</td>
</tr>
<tr>
<td>13</td>
<td>113</td>
<td></td>
<td>413</td>
<td>513</td>
</tr>
<tr>
<td>14</td>
<td>114</td>
<td></td>
<td>414</td>
<td>514</td>
</tr>
<tr>
<td>15</td>
<td>115</td>
<td></td>
<td>415</td>
<td>515</td>
</tr>
<tr>
<td>16</td>
<td>116</td>
<td></td>
<td>416</td>
<td>516</td>
</tr>
</tbody>
</table>

Fig. 4. Base index example: (a) eMAP-14: the shaded elements are the ones accessed by the whole 8 × 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).
and \( y \); thus, \( M \) should have at least one 1 per row.
- **2) Row**: eMAP needs to specify a dependency on \( t_x \) and no dependency on \( t_y \); thus, \( M \) should have no 1 on the bottom row, and at least one 1 on the top row.
- **3) Column**: eMAP needs to specify a dependency on \( t_y \) and no dependency on \( t_x \); thus, \( M \) should have no 1 on the top row, and at least one 1 on the bottom row.
- **4) Block**: similar to (1).
- **5) Neighbor**: similar to (1).

After removing the under/overspecified MAPs, only 34 MAPs remain valid, and are listed in Table III.

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.

IV. DESIGN SPACE AND CODE GENERATING

A. Benchmark Design Space

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

1) Local Space Allocation: OpenCL provides programmers with two ways to allocate local memory space. The first is to *statically* allocate the space in kernels with predefined size, and the second way is to *dynamically* allocate the space using APIs during run-time. We prefer the former approach when the size can be known in advance for better performance.

Regarding the size of local space, we propose two alternatives: (1) min-approach, and (2) max-approach. 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 Figure 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, and may enable more work-groups active, leading to better performance (Their performance comparison is shown in Section V-C). However, when using the min-approach, programmers need to perform work-item binding and data element shuffling. By comparison, the max-approach is easier for implementation.

2) Local Memory Access: 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 fall into different banks. For now, by using padding data, we manually remove bank-conflicts from the generated microbenchmarks, as a post-optimization step for MAPs like MAP-204.

B. Generating Benchmarks

In general, when using local memory, each microbenchmark consists of three main steps: allocate local memory space, load data elements into local memory, and use them. Taking MAP-407 (Figure 5) for example, we show the three steps in details.

**Step I: Allocating local memory**

As shown in Figure 5, the min-approach and max-approach need different amount 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, \( R \) is the radius of the block \((R = R_x = R_y)\). In Figure 5, \( WG = 8 \) and \( R = 1 \). Thus, the min-approach needs 50 cells, while the max-approach needs 100 cells.

**Step II: Loading data into local memory**

When moving data elements from global memory to local memory, multiple passes are needed. With the max-approach, we can bind one thread to one data element in the central shaded area (outlined by dashed square in Figure 5(c)) and mask the other threads. Thereafter, we need to load the border data into the local space. Nevertheless, loading data with the min-approach is complicated. Apart from thread masking, we have to deal with the *data shuffling* to put the data elements in the right places.

**Step III: Accessing local memory**

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 Figure 5(b) and 5(c)). Besides, we need to re-shuffle the access index when using the min-approach.

In this way, we can generate 68 \((34 + 34)\) kernels and 3190 lines of kernel code.

V. PERFORMANCE DATABASE

A. 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. We run each kernel \( R \) times and measure the total execution time \( T \). Then we calculate the bandwidth as \( W \times H \times N \times R \times \frac{size(type)}{T} \). 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 \( \frac{mbr = B}{b} \). If \( mbr > 1 \), using local memory is beneficial in terms of memory bandwidth; otherwise, using local memory leads to a performance loss.
B. Experimental Setup

We have run and compared the benchmarks on four platforms, whose configurations are shown in Table IV. When measuring the performance, we used six data sets \((dh \times dw)\) in Section III): 128 \(\times\) 128, 256 \(\times\) 256, 512 \(\times\) 512, 1024 \(\times\) 1024, 2048 \(\times\) 2048, 4096 \(\times\) 4096. Note that the parallelization of our microbenchmarks is based on output data, and thus \(dh = H\) and \(dw = W\). For the Block MAPs, we set the radius to be 3 \((R_x = R_y = 3)\) and we expect a larger memory bandwidth increase with a larger radius. Furthermore, we note that for a few cases (e.g., MAP-401) some data sets took too much time and their executions were aborted. In these cases, we only report the results for the data sets that finished within 2 hours. We set the work-group size to be 16 \(\times\) 16 (commonly used).

C. Performance Comparison of the max/min Approaches

![Fig. 6. Performance comparison of the max/min approaches.](image)

For MAP-407, we compare the \(mbr\) of the max and min approaches on the four platforms in Figure 6. 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 (up to 2\(\times\) faster) on GTX280. On HD6970 and GTX580, the performance of the min-approach is slightly worse. We also note that the bandwidth suffers around 70% loss with the min-approach on E5620. 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. In the paper, we record both of them in our performance database and show the larger \(mbr\) in the Figure 8.

D. Performance Database

1) Database Entry: After running the microbenchmarks, we get the performance database indexed by three items (platform, map, dataset) shown in Figure 7. Once the index is specified, a query in the database will return a database entry. Each entry consists of the memory bandwidth without local memory \((b)\), the memory bandwidth with local memory \((B)\), and their ratio \((mbr)\).

![Fig. 7. The database dimensions and its entry.](image)

2) Observations: For demonstration simplicity, we only show the \(mbr\) of the performance database in Figure 8: the horizontal axis represents the six data sets and the vertical axis represents \(mbr\). Overall, we found that the performance benefits of using local memory are heavily dependent on the size of data sets. Therefore, for a given (MAP,platform) pair, the user needs to select a right granularity to decompose the whole data sets. Besides, we make the following observations for each platform:

* **GTX280**: differs from other processors in that it has local memory, but no caches. We see that using local memory on GTX280 can achieve a memory bandwidth increase for most memory access patterns (32 out of 34 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 II, data reuse is a common indicator of using local memory. Taking the Single MAPs for example (see Table III), 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, there is a performance loss \((mbr < 1)\), 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 requirement from the off-chip memory can be met 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.

**HD6970 and GTX580:** have not only local memories but also caches. Similar to GTX280, using local memory on these two platforms is highly beneficial in most cases. However, the gain of memory bandwidth from using local memory is heavily dependent on the architecture. On GTX580, the bandwidths follow the same trends with that on GTX280, but the changes are less significant. Specifically, the older cacheless GTX280 benefits much more from local memory than the newer GTX580. 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 better use of the inherent data locality in the MAP, and therefore may perform better (see MAPs-109, 508, 509, 514). In other cases, with more complicated locality patterns, explicit usage of local memory remains beneficial on GTX580 (see MAPs-204, 303, 410, etc). For most MAPs on HD6970, the bandwidth first increases and then decreases over the data sets, which significantly differs from that on NVIDIA GPUs. We assume this is because HD6970 has a different cache architecture and implementation from that of GTX580.

**E5620:** has only caches on-chip, and implements local memory on global memory in an emulation mode. Thus, using local memory is equivalent to using the off-chip global memory, which might slow down the execution. However, we get better performance for MAPs-(107, 204, 205, 302, 303, 306, 401, 407, 408) by using local memory. We assume this is due to better caching or compiler optimizations. For other cases, using local memory degrades the memory bandwidth. When using local memory on E5620, we need to keep these beneficial MAPs in mind. We also note that using local memory on Intel Xeon X5650 (with larger caches than E5620) will enable a broader range of memory access patterns to have a bandwidth increase.

**3) Performance Gain/Loss Distribution:** We show the overall performance gain/loss distribution in Table V. We note that using local memory gives us a bandwidth increase for many MAPs (the ‘gain always’ percentage is 88% for GTX280, 68% for HD6970, 85% for GTX580, and 12% for E5620), and we recommend using local memory for such MAPs based on the guidelines mentioned in Section IV; 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 V**

<table>
<thead>
<tr>
<th>Performance Gain/Loss Distribution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX280</td>
</tr>
<tr>
<td>Gain Always</td>
</tr>
<tr>
<td>Loss Always</td>
</tr>
<tr>
<td>Varied</td>
</tr>
</tbody>
</table>

**4) 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 in Figure 9. We see that 12 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. Besides, there are 16 out of the 34 MAPs that can use both of them. Although data reuse over data sets can be quantified, quantifying the changes of global memory access will be difficult, thus performance unpredictability. Furthermore, the introduction of caches (HD6970 and GTX580) and the implementation differences of local memory (GPUs versus CPUs) make the performance more unpredictable (see Figure 8).

**VI. TOWARDS QUERY-BASED PERFORMANCE PREDICTION**

In this section, we illustrate how to predict the performance gain or loss from using local memory based on our performance database.

**A. Preliminaries**

Simply put, any kernel has two basic types of operations: computation and memory access. They are typically overlapped. Therefore, we can roughly estimate the kernel execution time as $T = \max(T_C, T_M)$, where $T_C$ and $T_M$ are the time spent by the kernel in computing and accessing the memory, respectively. If $T_C$ is larger, the kernel is compute-bound, while in the case $T_M$ is larger, the kernel is memory-bound.

When $T$ is dominated by $T_C$, using local memory brings no performance improvement [12]. When $T$ is dominated by $T_M$, using local memory is one of the recommended techniques to improve the overall performance [8] [7], by reducing $T_M$. Thus, the goal of our performance prediction method is to use the bandwidth values stored in the performance database to determine the improvement in access time when using local memory: $T_{LM}^{T_M} = T_M/mbr$. Using this, we can compute the overall execution time of the application as $T_{LM} = \max(T_C, T_{LM}^{T_M})$.

We note, however, an important issue: in case $T_{LM}^{T_M}$ becomes smaller than $T_C$, the kernel becomes compute-bound...
after the application of local memory, and therefore the new execution time will be dominated by the computation time. In this case, our prediction will always be optimistic (i.e., we will always predict a higher performance than the actual performance to be achieved) because we can only predict the increase in performance due to the decrease of $T_{bm}$. However, by providing this upper bound to the achievable performance, we can safely indicate that when $mbr_p \leq 1$, using local memory will not be beneficial, and thus not recommended. Alternatively, in the cases when $mbr > 1$, we recommend using the local memory, but the observed gain might be lower than expected. To alleviate this uncertainty, we are currently working on a model that could predict this shift (i.e., from memory-bound to compute-bound) and consequently adjust the prediction.

B. Case studies

We further exemplify our proposed performance prediction method with four applications: Matrix Transpose (MT), Matrix Multiplication (MM), Image Convolution (IC), and Successive Over-Relaxation (SOR) [13]. When predicting the memory bandwidth of using local memory, we first identify the MAPs from the applications. We note this step is, for now, beyond the scope of this work. Thus, for these case studies, we identify the MAPs by observing the algorithm and its implementation.

Once the MAPs are identified, a query in the performance database returns $b$, $B$, and $mbr$ for different dataset sizes. We use these values to derive the predicted memory bandwidth ratio ($mbr_p$) of using local memory on GTX580. We note here that not all possible values of dataset sizes are included in the database. We expect the user will have some freedom in choosing a suitable dataset size when implementing a local memory version of the kernel (and then we recommend the usage of the listed with the best performance). However, in case the choices are severely restricted, the user needs to choose one of these two solutions: either choose the closest data size and assume the prediction is able to indicate performance loss/gain, or rerun the benchmark with the required dataset size (thus, effectively extending the database).

To validate our query-based prediction, we fix the size of the dataset at $1024 \times 1024$ and we measure the execution time for these four kernels without and with local memory ($T$ and $T_{LM}^*$, respectively) on the same platform. We present our results in Table VI. When the kernel has only one input matrix and thus one memory access pattern, it is straightforward to derive $mbr_p$ by assigning $mbr$ of the identified MAP. MT and SOR are such cases. When looking into the code, we found that the MAPs of MT and SOR are MAP-110 and MAP-508, respectively. In Table VI, we see that using the database gives an accurate performance estimate for MT and SOR (within 5%). Thus, the query-based approach can predict the performance impacts with high precision for an isolated MAP.

When the kernel has more than one input matrix, the composition becomes complicated. In general, we calculate $mbr_p$ with Equation 4, where $\otimes$ and $\oplus$ represent application-dependent operators (typically multiplication, addition, average, or min/max) and $m$ is the number of different input matrices. In the following, we take IC and MM for example ($m = 2$) to illustrate how to get $mbr_p$.

$$mbr_p = \frac{B_1 \otimes B_2 \otimes \ldots \otimes B_m}{b_1 \oplus b_2 \oplus \ldots \oplus b_m}$$

(4)

For MM, we calculate the matrix multiplication by $C \leftarrow A \times B$. The memory access patterns of $A$ and $B$ resemble MAP-205 and MAP-302, respectively. We suppose the data amounts of off-chip data transferred are $n_A$ and $n_B$ for Matrix A and Matrix B ($n_A = n_B$). Without using local memory, Matrix A and Matrix B will compete for caches and the overall bandwidth will be smaller than the case when A or B uses caches separately. Thus, we use the smaller memory bandwidth $\min(b_A, b_B)$ as the bandwidth without local memory. We calculate the predicted $mbr = 2.40$ of using local memory in Equation 5.

$$mbr_p = \frac{(n_A \times B_A + n_B \times B_B)}{\min(b_A, b_B)}$$

(5)

For IC, we have two input matrices (the input image $IM$ and the filter) each with a different memory access pattern. The memory access pattern of $IM$ is the Block MAP-408. We set the radius to be $7 \times 7$, and thus the filter size will be $7 \times 7$. Note that the filter data is allocated on the constant memory space. Since the filter is small, it can be loaded into caches once and then be reused/shared among all threads. Thus, the access time of the filter can be ignored, and we use the $mbr$ of MAP-408 to approximate $mbr_p$ ($mbr_p = 1.83$). However, we note a larger error margin for IC in Table VI. In this case, we see that our prediction is too optimistic when compared with the measured performance. This is due to the shift of the kernel from being memory-bound to being compute-bound. As the new kernel (i.e., the one that uses local memory) is compute-bound, the performance we see in Table VI is actually limited by the computation, not by the bandwidth, and therefore our prediction is inaccurate. As already mentioned, this inaccuracy can be signaled by coupling the query-based prediction with a simple model to predict this kernel shift (using, for example, the Roofline method).

To summarize, we have shown how to predict benefits of using local memory on four examples. We see that the query-based performance prediction is a promising approach. We are currently working on model-based detection of kernel memory-to-compute bounds. In the near future, we will focus on studying interference of memory access patterns, aiming to generalize them in a model to support accurate multi-MAP performance impact of local memory usage.

VII. Related Work

In this section, we discuss prior work on benefits of prediction and code transformation for local memory. In [3], the author presents a method of computing precisely which memory cells are reused due to temporal locality. In [4],
Isshin presents an automated approach for analyzing the data reuse opportunities in a program, allowing efficient use of local memory. The approach can reduce energy consumption and improve performance. However, they focus on enabling local memory only when data reuse is available. Our study tackles a more generic problem.

Research on code transformer enabling local memory is yet another interesting topic. Baskaran et al. [14] develop an approach for effective automatic data management for on-chip memories, including creation of buffers 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 [15], Yang et al. propose a GPGPU compiler, which converts the un-coalesced accesses to coalesced ones, and enhances data reuse. Our work is orthogonal to this work, as our solution is intended to estimate the performance gain of their approaches.

A more generic related topic is auto-tuning. Generally, there are two types auto-tuning: empirical optimization [16] [17] [18] and model-driven optimization [19] [20] [21]. Although empirical techniques giving the optimal performance, 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, but it may not give optimal performance, because analytical models are simplified abstractions of architectures and/or applications. Our work relates to both: we use modeling to build our database, and use the database to potentially prune the search space of empirical auto-tuners.

**VIII. 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 paper, we presented a benchmark-based approach to tackle this issue starting with the memory access patterns (MAPs) of many-threaded applications. 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 NVIDIA GPUs (GTX280, GTX580), AMD GPUs (HD6970), and Intel CPUs (E5620), and obtained a performance database. 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.

For future work, we plan to investigate more MAPs and platforms using our methodology, providing a more extensive database. As the query-based performance prediction has shown promising results, we also plan to investigate the MAP interferences. Finally, we want to implement an auto-tuner (of using local memory) based on the performance database.

**Acknowledgment**

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

**References**

Fig. 8. Performance database for 6 datasets expressed as \( mbr \) (memory bandwidth ratio) between not using and using local memory: the horizontal axis represents the six data sets and the vertical axis represents \( mbr \). The dashed line represents \( mbr = 1 \) (no performance gain or loss), the thick-solid line represents \( mbr = 0 \), the thin-solid (red) line represents \( mbr \) for different datasets, and the dots above the line (\( mbr = 1 \)) stand for better performance using local memory. We also mark the maximum range of \( mbr \) in each sub-figure.