LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs

Devashree Tripathy[‡], AmirAli Abdolrashidi[‡], Quan Fan[‡], Daniel Wong[‡] and Manoranjan Satpathy[¶]
[‡]University of California, Riverside, CA, USA
[¶]Indian Institute of Technology Bhubaneswar, Odisha, India
{dtrip003, aabdo001, qfan005, danwong}@ucr.edu and manoranjan@iitbbs.ac.in

***Abstract*—Exploiting data locality in GPGPUs is critical for efficiently using the smaller data caches and handling the memory bottleneck problem. This paper proposes a thread block-centric locality analysis, which identifies the locality among the thread blocks (TBs) in terms of a number of common data references. In *LocalityGuru*, we seek to employ a detailed just-in-time (JIT) compilation analysis of the static memory accesses in the source code and derive the mapping between the threads and data indices at kernel-launch-time. Our locality analysis technique can be employed at multiple granularities such as threads, warps, and thread blocks in a GPU Kernel. This information can be leveraged to help make smarter decisions for locality-aware data-partition, memory page data placement, cache management, and scheduling in single-GPU and multi-GPU systems.**

The results of the LocalityGuru PTX analyzer are then validated by comparing with the Locality graph obtained through profiling. Since the entire analysis is carried out by the compiler before the kernel launch time, it does not introduce any timing overhead to the kernel execution time.

***Index Terms*—GPGPU, Cache management, Data locality, Syntax tree**

I. Introduction

General-purpose graphics processing units (GPGPUs) have evolved as widely used accelerators delivering high performance and throughput to many important application domains like scientific computing, graph processing and machine learning [1]. However, the modern day GPGPUs suffer from memory contention, lack of parallelism and load imbalancing resulting in under-utilization and energy inefficiency [2]–[4]. Hence, efficient use of the memory system as well as co-locating the compute and data together is important for exploiting the massive computational capability offered by GPGPUs to their full potential. A key approach to efficiently improve the memory bandwidth is *data locality* – (i) increasing the data reuse within the SM at the thread, warp and thread block (TB) levels, thereby reusing the L1 cache lines before it is evicted; and (ii) placing the data close to the computation so as to reduce the communication across multiple SMs within a GPU or even multiple GPUs. There is no generalized technique which exploits the data locality present in various applications to improve the efficiency of cache and memory system usage. There have been prior research on addressing the memory bottleneck issues, including prefetching [5], [6], cache management [7], locality-aware schedulers [8], [9], memory-level parallelism [10]–[12] etc. However, all these techniques need the data locality information which is either known a priori using profiling techniques, or learnt and predicted during the kernel execution [13].

Limitations of Prior Works: Some of the recent works use programming language constructs to associate a thread to the mapped data in order to extract the data address range accessed by the thread [14]–[17]. However, these approaches add to the programmer's burden of learning a new language and using it to rewrite the program. Locality Descriptor [18] expresses the data locality using program semantics, which needs the programmer to explicitly specify the static tile dimensions, compute and data mapping and data sharing pattern. CODA [19] uses static analysis to compute the stride distance between two consecutive thread blocks at runtime. This information is used to determine the size of the data accessed by a TB and ensure that the TBs and the data they access are co-located on the same GPU. Though this approach captures the TB index range in case of the 2D regular grid applications exhibiting strided accesses, a more detailed analysis of the code is needed to account for other access patterns in various applications. TAFE [20] estimates the thread address footprints of the static as well as dynamic data-dependent applications before kernel launch. The thread to data address index range relation coefficients are extracted by manual inspection of the application source code. LADM [21] classifies the TB locality pattern in the application into one of seven patterns using static compiler analysis to check for loop variance of array indices. However, this process uses the *CUDA source code*, which may not be always available to the user, whereas LocalityGuru seeks to find the relationships between the GPU registers through PTX analysis.

The static analysis helps us determine the stride distance between two consecutive thread blocks in the grid as well as the data-dependency between the TBs due to the loop iterations. In LocalityGuru, we perform a detailed static compiler analysis to automatically extract the thread to index range relationship from the intermediate representation (IR)

978-1-7281-7744-1/21/$31.00 © 2021 IEEE

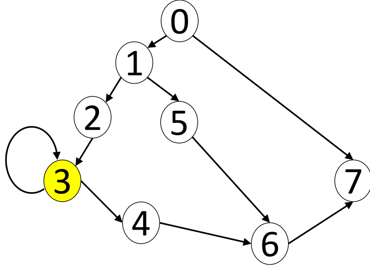

Fig. 1. Control flow graph of the PTX basic blocks (bb) for matrix multiplication. Basic block 3 (highlighted) contains the `ld.global` instructions and has a self-loop.

of the source code (in PTX format). This paper makes the following contributions:

- We analyze the PTX code at JIT compilation time before the kernel launch and perform the detailed static index analysis to derive the equation for the thread/TB mapping to data element indices accessed.
- We validate the results of the TB locality graph obtained through automated LocalityGuru PTX analyzer by comparing with the profiling data-locality results. Our approach imposes *zero timing overhead* on the kernel execution time.

The paper is organized as follows: Section II describes the PTX analysis background. Section III discusses the process of constructing the syntax trees and using them to extract the locality behaviour of the kernel at the Thread-Block (TB) level. Our results are discussed in Section IV. We describe related work in Section V and we conclude in Section VI.

II. Background

Static analysis for a GPU application has to take into account the parallel nature of thread block executions and data accesses. Otherwise, it is little different from a static analysis meant for a sequential application.

A benefit of analyzing a flat IR such as PTX code over the CUDA source code is that it has fewer program constructs and simpler program semantics. For example, CUDA constructs such as “for loop” and “while loop” that are syntactically different, but semantically equivalent, tend to correspond to similar PTX. Similarly, the conditional “if-else” and “ternary” statements that have *syntactically different* but *semantically equivalent* CUDA source code are compiled to similar PTX code. PTX representation uses an assembly-like structure for a virtual GPU architecture, and therefore is at a much more abstract level than the SASS format which executes on a specific GPU architecture.

Target architecture-independent PTX programs have an assembly-language-style syntax with instruction operation codes (opcodes) and operands. A common way of representing PTX code is through the use of control flow graphs (CFGs). Example of CFG for matrix multiplication is shown in Figure 1. Nodes of a CFG represent basic

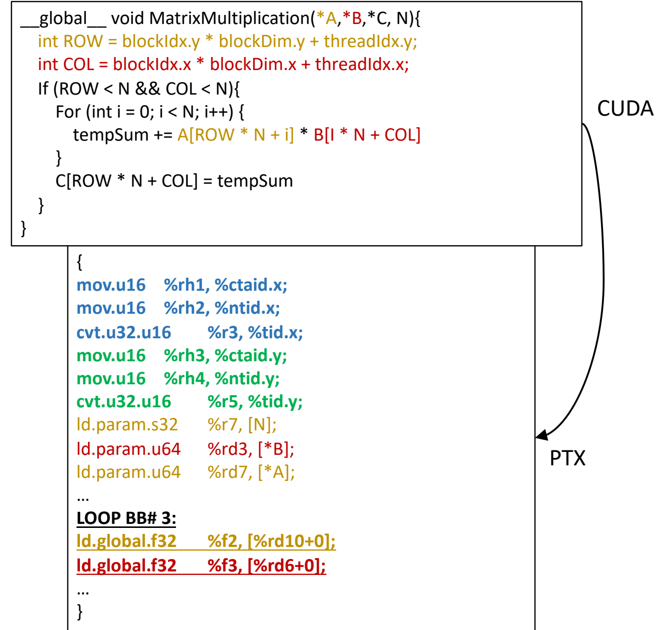

Fig. 2. CUDA source code and PTX IR for for Matrix Multiplication Application

blocks, which are sequences of instructions without any control flow statements in between. Edges are directed from a source basic block to a target basic block showing the possibility for control to jump from the source block to the target block. There can be up to two possible edges from a given basic block (in case of a conditional jump, leading to a *true block* and a *false block*.) A great deal of information can be derived from the CFG structure, including the registers accessing a certain memory block, their order, and sometimes the frequency, of those accesses. In this work, we aim to extract this information before runtime in order to help with the GPU’s cache management and locality-aware task scheduling.

III. LocalityGuru

This section discusses a PTX analysis based approach to determine the locality of threads, warps and TBs at just-in-time (JIT) compilation time using syntax trees.

A. PTX Analysis with Syntax Trees

Abstract syntax trees (referred to as *syntax trees* in the paper) are used to represent the syntactic structure of PTX code. The trees of the programming constructs like arithmetic and logical expressions, and control flow statements are grouped into operators (root nodes) and operands (leaves of the root nodes). For example, the syntax for instructions in PTX are as follows:

$$opcode.type \quad dst, src1, src2[, src3]$$

where $.type$ represents the type of the source operands. ($.type \in \{.u16, .u32, .u64, .s16, .s32, .s64\}$). The syntax tree

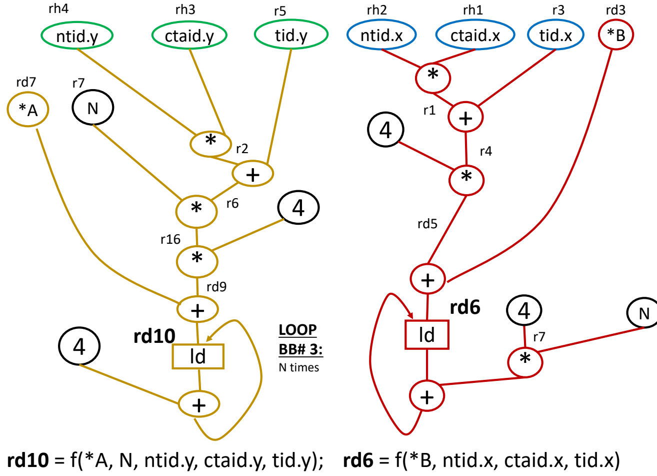

Fig. 3. Abstract Syntax for Matrix Multiplication Application

Algorithm 1: Syntax Trees for global load registers

```
   // ST:Syntax Tree
1  function BuildSyntaxTree(I, ST)
2      if dst(I) in S then
3          node ← leaf_node_in_ST
4          node.opcode ← I.opcode
5          Erase dst(I) from S
6          node.left ← create_new_node(I.src1)
7          node.right ← create_new_node(I.src2)
8          node.third ← create_new_node(I.src3)
9      end
10 function TraceBasicBlocks(BB, ST)
11     if BB has predecessors then
12         for BB_j in BB_predecessors do
13             for I in BB_j_inst_reversed do
14                 BuildSyntaxTree(I, ST)
15             end
16             TraceBasicBlocks(BB_j, ST)
17         end
18     end
   // The main function
19 function GetAllSyntaxTrees(PTX_instructions)
20     for I in PTX_instructions do
21         if I.opcode is ld.global then
               // ld_reg = src(I)
               // Initialize syntax tree for load
                  register: ld_reg
22             ST ← create_new_node(ld_reg)
23             Insert ld_reg into S
24             for I_i in BB_inst_reversed do
                   // BB is the basic block
                      containing global load
                      Instruction
25                 BuildSyntaxTree(I_i, ST)
26             end
27             TraceBasicBlocks(BB, ST)
28         end
29     end
30     return ST
```

expression for the instruction will have the opcode as the root node, destination operand (*dst*) as the result, and source operands 1 and 2 ($src1$ and $src2$) as the left and right child nodes respectively. Note that PTX also supports a third source operand for some operations, such as multiply-and-add (`mad_op`), in which $dst = (src1 * src2) + src3$. The construction of the syntax trees is explained in Section III-B.

Figure 2 shows the example of the translation of the CUDA source code into the PTX intermediate representation in the matrix multiplication application. The CUDA source code is converted to its PTX representation during the offline compilation phase, and the kernel parameters are ready just before the kernel launch during the JIT compilation. After the kernel arguments are known, PTX is converted to architecture-specific SASS for execution. The advantage of doing locality analysis at PTX instead of SASS is that PTX is hardware agnostic and has added information from the kernel parameters. LocalityGuru works at the JIT compilation time to extract the thread-level address-data footprint using the abstract syntax tree. In some applications, the kernel parameters such as the `GridDim`, `BlockDim` and input data size `N` are dependent on the user input and are determined only at JIT-compilation time. In matrix multiplication, the exact values of the matrix size `N` and the base addresses of input/output matrices `A`, `B`, `C` are known after `malloc` in GPU. When the grid size and TB size become available, so do the ranges of the existing thread-specific values, namely `threadIdx` and `blockIdx`. The load address range accessed per thread is stored in the source operand of the global memory read instructions, e.g., `ld.global`. The control flow graph (CFG) derived from matrix multiplication's PTX is shown in Figure 1. We start from the basic block with the `ld.global` instruction (BB 3) and recursively parse the instructions in all its predecessors until the leaf nodes consist of immediate values and kernel parameters only.

Some of the corresponding codes in PTX (Figure 2) and syntax tree (Figure 3) are highlighted using matching colors. The statements used for calculating the syntax tree of matrix A are in brown color and the matrix B are in red color across CUDA , PTX as well as syntax tree edge and node color annotations. After locating the global memory access instructions, an abstract syntax graph (ASG) is constructed for each `ld.global` source register (`rd10` and `rd6` from Figure 3) to identify which elements of the arrays (`A` and `B`) are accessed by each thread, thereby determining the value range of the memory accesses per thread/TB. In the ASG, the leaf nodes are the registers storing the values known at kernel-launch time, such as `&A`, `&B`, `N`, `GridDim` and `BlockDim`. Using the syntax tree, `rd10` is expressed as a function of `&A`, `N`, `ntid.y` (same as `BlockDim.y`), `ctaid.y` and `tid.y` i.e. $\texttt{rd10} = \texttt{\&A} + [ROW * \texttt{N} + i]$ in CUDA. The kernel parameters in `.x` and `.y` dimension are in blue and green respectively in the PTX (Figure 2) and comprise of the leaf nodes in syntax tree (Figure 3) of `rd6` and `rd10` respectively.

Using this, we can obtain the common set of elements

in each matrix accessed by every TB, e.g., A_i, for TB_i, etc. Therefore, the number of common data elements accessed by TB_i and TB_j in the locality graph (Figure 4 - matrix multiplication) will be:

$$L(TB_i, TB_j) = |A_i \cap A_j| + |B_i \cap B_j|$$

B. Syntax Tree Construction

Algorithm 2: Locality Graph from Syntax Tree

```
   // ST:Syntax Tree
   // ST_loop:Syntax Tree for loop_variable
   // L:Locality Graph for TB
   // Map: TB_Address_map
   // reg: global load register
1  function EvalST_loop (ST,Map)
2    if loop in reg.BB then
3      while EvalST(BB_label) != 0 do
4        Update loop_iterator in ST(BB_label)
5        Update leaf_node reg in ST_loop(reg)
6        Map[TB].insert(EvalST(reg))
7      end
8    end
9  function GetTBAddressMap(ST)
     // Assign kernel parameters to leaf nodes
        in ST
10   ntid ← BlockDim
11   forall ctaid in GridDim do
12     TB ← get_tb_id(ctaid,GridDim)
13     forall tid in BlockDim do
         // Filter out idle threads in False
            branch BB
14       for P not in BB_False_branch do
           // P: Predicate
15         if ! EvalST(P) then
16           forall offset(reg) do
17             Insert (EvalST(reg)+offset) into
                 Map[TB] // BB with loops
18             EvalST_loop(ST,Map)
19           end
20         end
21       end
22     end
23   end
24   return Map
   // The main function
25 function GetLocalityGraph()
26   Map ← GetTBAddressMap(ST)
27   forall ctaid in GridDim do
28     if TB_i != TB_j then
29       L[TB_i][TB_j] = Map[TB_i] ∩ Map[TB_j]
30     end
31   end
32   return L
```

Algorithm 1 shows the pseudo-code of the syntax tree for global load source registers. We locate the global load source registers in the PTX code by tracing the `ld.global` instructions and constructing a syntax tree (ST) for each (lines 19-30). For each instruction I_i in the basic block containing register *ld_reg*, `BuildSyntaxTree()` looks back for the instruction with *ld_reg* as its result. A node is then created for I_i and inserted into ST (lines 1-9). After iterating through a BB, all its predecessors are iterated recursively (with no loops) to trace the source of the register using `TraceBasicBlocks()`, and the tree is updated accordingly (lines 10-18).

Handling Branches:

Every basic block having more than one successor ends with a branch instruction. The true branch leads to BB 3, while the false branch avoids it. A conditional branch instruction in PTX uses a predicate to dictate whether it should execute the following basic block. During the locality graph construction shown in Algorithm 2 (Line 14), if the predicate value points to the false branch for a thread, it is removed from the locality calculations.

In many cases, not all the threads are assigned a task due to the input data size not being a multiple of number of threads, resulting in some threads in the last row or column of the grid idling throughout the kernel (*idle threads*). Thus, while generating the locality graph, the idle threads need to be filtered out in the analysis. In our matrix multiplication example, the threads executing BB 5 and BB 7 are idle threads (Figure 1). In order to determine whether to skip idle threads, we should build syntax trees for the predicates in the predecessor basic blocks to identify the threads executing the false branch basic blocks.

Handling Loops:

In our matrix multiplication example, BB 3 has a self-loop (i.e. a branch in which the predecessor is same as successor) which points back to its beginning (Figure 1). The loop has an iterator which is incremented by 1 in every loop iteration and ends the loop when it reaches the value of `N`. In every loop iteration, `rd6` and `rd10` represent $\&A + [ROW * N + i]$ and $\&B + [i * N + COL]$ respectively where i is the loop iteration count. A separate syntax tree (ST_{loop}) is constructed per loop variable which captures the formula for updating the loop variables in each iteration. We evaluate the syntax tree for the loop as in Algorithm 2 `EvalST`$_{loop}$`()` (Line 1), where the leaf nodes of the syntax tree are updated with the values of the loop variables in the previous iteration, and then evaluate the syntax tree result for the predicate at the end of the loop to decide whether to continue to the next loop iteration by branching to its start, or exit the loop, as shown in Algorithm 2 (Line 3).

Handling Address offsets and data-hazards in the source register:

Multiple global load instructions may use the same source register with different address offsets. In this case, we look for any data hazard due to register write into that register between the two instructions. If there is a data hazard, then separate syntax trees are constructed for it. Otherwise, we construct only one syntax tree, and store the register name and address offset in a map. During the syntax tree evaluation phase in Algorithm 2 (Line 16), the address offset

is added to the calculated value of the syntax tree for the register.

C. Locality Graph from Syntax Tree

Algorithm 2 shows the pseudo-code of obtaining the locality graph from the syntax tree. The STs constructed in Algorithm 1 are used to map thread block ID to memory addresses it has accessed (*TB_Address_map*) (lines 9-24). Since we already represented the global load source registers in terms of kernel parameters in the syntax tree, a set of memory addresses can be calculated for every thread ID and TB ID. All the cases of *idle threads*, address offsets in the source register, and loops are taken into account while evaluating the syntax tree to get the address range as described in Section III-B. Once *TB_Address_map* is populated with values for all TBs and all global source registers, the locality graph L is constructed for each kernel by intersecting the read memory address sets of TB pairs (lines 25-32).

The syntax trees are constructed per PTX file. However, since each kernel can have different program body and kernel parameters, the locality graph is constructed per kernel which also accounts for the changed input data per kernel.

D. Summary:

The syntax tree derives the thread-to-memory-addresses-accessed relationship in terms of thread ID, block ID and other kernel parameters. This information can be used to capture inter-thread, inter-warp, inter-TB locality within the same kernel as well as across multiple kernels. Though we have shown an example to capture the locality due to memory read of the same data, LocalityGuru can also be used for data-dependency analysis among TBs in multiple kernels. In that case, each TB shall have a *TB_Address_map* for both read and write accesses. This inter-kernel TB-level data dependency (or producer-consumer relationships) can be used for hardware optimization techniques like TB scheduling as discussed in [8], [22], [23]. At the intra-kernel level, LocalityGuru can aid in the optimization techniques proposed in [9], [24]–[28].

The applications evaluated in this paper (Section IV) contain thread ID-dependent accesses. Our technique can also be extended to be used for the indirect accesses where the result of one memory access (thread ID-dependent primary access) is used to calculate the address of the next memory access (secondary access). The primary data structure can be passed to the PTX analyzer as an argument to extract the access patterns [20], [29].

IV. Results and Discussion

Methodology: We implement and perform the PTX analysis for thread-level address footprint using the built-in PTX parser in GPGPU-Sim [30]. Our analysis algorithm is generic and can be implemented in any compiler framework that supports PTX, such as LLVM [31] and GPUOcelot [32]. It is fully automated to do the locality analysis on any unknown application in CUDA or OpenCL which generates PTX. In our experiments, 12 benchmarks used were selected from 3 different benchmark suites NVIDIA CUDA SDK [33] (Matrix Multiplication), Rodinia [34] (Hotspot) and Polybench [35] (SYRK, SYR2K, 2MM, GEMM, BICG, Covariance, Convolution 3D/2D, MVT, Gram-Schmidt).

Understanding the Patterns: The terminology used in our discussion is as follows:
bx: `blockIdx.x`, *by*: `blockIdx.y`,
tx: `threadIdx.x`, *ty*: `threadIdx.y`,
$index_x$: (*bx* * `blockDim.x` + *tx*),
$index_y$: (*by* * `blockDim.y` + *ty*)
If a global load address is only dependent on $index_x$ *or* $index_y$, it is referred to as an $index_x$-only or $index_y$-only pattern here. When the address is dependent on $index_x$ *and* $index_y$, it is referred to as $index_{xy}$ pattern. When the address is dependent on $c_x index_x + c_y index_y$, it is referred to as $index_{x+y}$ pattern, where c_i, c_j are constants.

In matrix multiplication, for input size of 200, `GridDim`=13x13 and `BlockDim`=16x16, the accesses to input matrices are $\texttt{A}[index_y * 200 + i]$ and $\texttt{B}[i * 200 + index_x]$ where $i \in \{0, ..., 199\}$. Therefore 13 consecutive TBs in the same column of the TB grid share $16 \times 200 = 3200$ data elements (shown as the triangles in the diagonal of the matrix multiplication in Figure 4). Similarly, the TBs in the same row of the TB grid share 3200 data elements, which is shown as accesses by a strided distance of 13 (i.e. `GridDim.x`) in the figure.

In 2MM, for input size `N` of 256, `GridDim`=8x32 and `BlockDim`=32x8, the read accesses to matrices are $\texttt{A}[index_y * \texttt{N} + (0...\texttt{N}-1)]$, $\texttt{B}[(0...\texttt{N}-1) * \texttt{N} + index_x]$ and $\texttt{C}[index_x * \texttt{N} + index_y]$. The TBs in the same row of the TB grid exhibit $index_y$-only pattern and share $\texttt{blockDim.y} * \texttt{N} = 2048$ elements and in the same column exhibit $index_x$-only pattern and share $\texttt{blockDim.x} * \texttt{N} = 8192$ elements. No sharing is observed for matrix `C` which has $index_{x+y}$.

Similarly, in GEMM, for input size `N` of 64, `GridDim`=2x8 and `BlockDim`=32x8, as the matrices `A` and `B` have similar access patterns as 2MM and matrix multiplication, row-wise sharing among the TBs is $\texttt{blockDim.y} * \texttt{N} = 512$ elements and in the same column share $\texttt{blockDim.x} * \texttt{N} = 2048$ elements.

In case of Gram-Schmidt, kernels 2/5/8 have `GridDim`=4 and `BlockDim`=256, and accesses input matrices $\texttt{a}[index_x * \texttt{N} + \texttt{k}]$ and $\texttt{r}[\texttt{k} * \texttt{N} + \texttt{k}]$ where `k` is a kernel paremeter. Here as matrix `r` indices are not a function of *bx* or *by*, it would be accessed by all the TBs and an $index_x$-only pattern of sharing would be observed. Now since the TBs are arranged

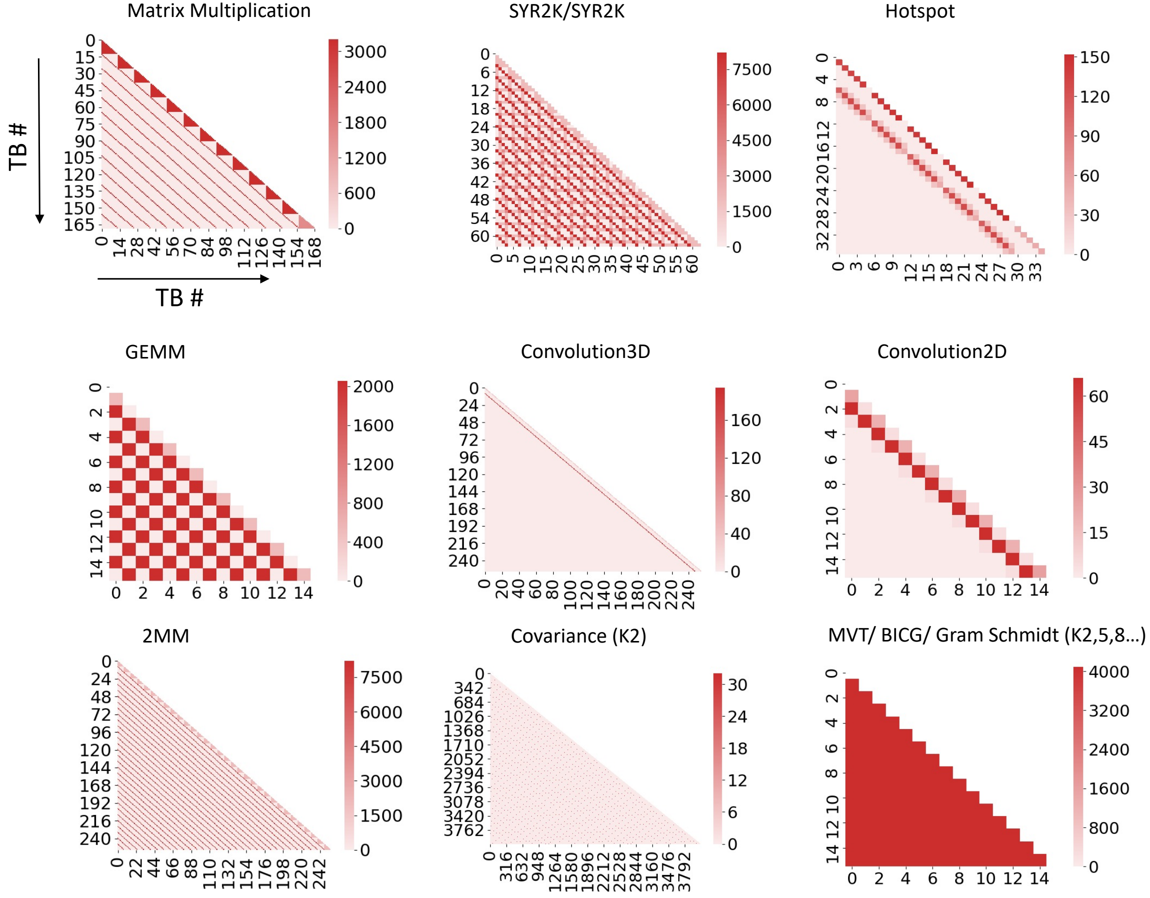

Fig. 4. TB locality graph results for different applications. The numbers in brackets represent the Kernel #. The adjacency matrix representation of locality graph is symmetric and has been shown as a lower triangular matrix. The TB # are shown in the x axis (increasing order) and y axis (decreasing order). The number of common memory addresses accessed by any two TBs is shown as the red color intensity in the heatmap. The more intense red color refers to more data-locality among the TBs.

in a 1D grid, we observe that the locality graph is fully connected i.e. every TB is connected to rest of the TBs in the kernel. Similarly, MVT has 1D kernel and accesses matrices $\texttt{x1}[index_x]$, $\texttt{a}[index_x * \texttt{N} + 0 ... \texttt{N} - 1]$ and $\texttt{y}_1[0...\texttt{N}]$. $index_x$-only pattern sharing in a 1D Kernel (along the x-axis) in applications like Gram-Schmidt, MVT and BICG results in fully connected locality graph.

In case of SYR2K, for input size `N` of 256, `GridDim`=8x32 and `BlockDim`=32x8, the read accesses to matrices are $\texttt{A}[index_y * \texttt{N} + i]$, $\texttt{A}[index_x * \texttt{N} + i]$, $\texttt{B}[index_x * \texttt{N} + i]$, $\texttt{B}[index_y * \texttt{N} + i]$ and $\texttt{C}[index_x * \texttt{N} + index_y]$, where $i \in \{0, ..., \texttt{N} - 1\}$. Here, the sharing pattern for both the matrices A and B is $index_{xy}$. No sharing is observed for matrix `C`. Since there are some common elements from $index_x \cap index_y$, we observe a different pattern for $index_{xy}$ as compared to $index_x$ only or $index_y$ only.

In case of Convolution2D, the kernel is 2D with `GridDim`=2x8 and `BlockDim`=32x8 and the read accesses to matrices are $\texttt{A}[(index_x + i) * \texttt{N} + index_y + j]$ where $i, j \in \{-1, 0, 1\}$. It has a stencil computation behavior where every pixel computation involves read accesses to all its neighboring pixels in xy-plane. Hence we observe that every TB shares elements with its neighboring TBs in both dimensions and has $index_{x+y}$ pattern.

In Covariance, kernel 2 is 2D with `GridDim`=64x64 and `BlockDim`=32x8, the read accesses to matrices are $\texttt{mean}[index_x + 1]$ and $\texttt{data}[index_y * (\texttt{N} + 1) + index_x]$, where $i \in \{0, ..., \texttt{N}\}$. We observe the TB-to-data mapping is in column-major order, hence the slope is reversed compared to the row-wise data mapping locality pattern seen in MM, GEMM and 2MM.

Validating the Results: In order to validate the results of LocalityGuru, the locality graphs were constructed for all kernels using profiling by running benchmarks in GPGPU-Sim and recording the memory addresses accessed by each

TB. We perform an element-wise comparison between the resulting locality graph from LocalityGuru and the simulator's generated locality graph. No differences were noted.

V. Related Work

Prior works have proposed various methods to improve data locality in different applications. In some works, the programmer provides hints in the code which would be used to optimize locality [18], [20], [29], [36]–[38]. Locality Descriptor [18] lets the user express and use the data locality information in GPUs through software to allow optimization for the programmer, combined with hardware to leverage the data locality in the application. Sometimes, new programming languages have been proposed to support the expression of data locality [14]–[17], [39]–[41]. However, LocalityGuru aims to minimize the programmer burden to either rewrite the code in a new abstraction or even add compiler directives to the already written code.

Compiler analysis has already been used by several works for data locality [21], [42]–[44]. Index analysis in compilers has been utilized to perform loop transformations in the source code targetted at improving data locality [45]–[47]. However, in this work, we choose to utilize the PTX intermediate code which is architecture-agnostic and holds the control and data flow information better than source code, which may not be always available to the user.

TAFE [20] allows the programmer to send static kernel data and dynamic memory information to the device in order to extract their data locality information, and contains hardware to track data-dependent accesses, reducing the software overhead. Therefore, indirect memory accesses can be obtained, albeit through user APIs in the code.

TB scheduling has also been used to improve data locality in GPUs. LaPerm [23] uses TB scheduling hardware for dynamic parallelism execution to improve cache performance by binding child TBs to the hardware resources used by their parent TB. Li et al. [26] employs a CTA clustering technique in order to maximize inter-CTA data reuse. These methods would be able to work orthogonally with our proposal in order to further improve their performance.

VI. Conclusion

In this paper, we aim to derive the relationship between the thread blocks and the memory addresses accessed by them. A detailed compiler analysis is performed on the PTX intermediate representation using syntax trees to extract the data locality in terms of the number of common data elements shared between all thread block pairs in a kernel. Our locality analysis technique can be employed at multiple granularities such thread-, warp- or TB-level in a GPU kernel. This information can be leveraged to help make optimizations for locality-aware data-partition, memory page data placement, prefetching, cache management and TB as well as warp scheduling in single or multi GPUs.

Acknowledgments

We also extend our gratitude to Dr. Laxmi Bhuyan for providing inputs at the various stages of the project. This work is partly supported by National Science Foundation under Grants NSF - 1907401, 1815643 as well as Govt of India SPARC project P712. We would also like to thank the anonymous reviewers for their invaluable comments and suggestions.

References

[1] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

[2] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 109–114.

[3] H. Zamani, Y. Liu, D. Tripathy, L. Bhuyan, and Z. Chen, "Greenmm: energy efficient gpu matrix multiplication through undervolting," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 308–318.

[4] H. Zamani, D. Tripathy, L. Bhuyan, and Z. Chen, "Saou: Safe adaptive overclocking and undervolting for energy-efficient gpu computing," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 205–210.

[5] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 73–82.

[6] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "Cta-aware prefetching and scheduling for gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 137–148.

[7] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "Dacache: Memory divergence-aware gpu cache management," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 89–98.

[8] A. Abdolrashidi, H. A. Esfeden, A. Jahanshahi, K. Singh, N. Abu-Ghazaleh, and D. Wong, "Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems."

[9] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[10] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "Dream: Dynamic re-arrangement of address mapping to improve the performance of drams," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 362–373.

[11] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 32–41.

[12] J. H. Zurawski, J. E. Murray, and P. J. Lemmon, "The design and verification of the alphastation 600 5-series workstation," *Digital Technical Journal*, vol. 7, no. 1, p. 0, 1995.

[13] L. Zhou, L. N. Bhuyan, and K. Ramakrishnan, "Goldilocks: Adaptive resource provisioning in containerized data centers," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 666–677.

[14] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for hpc with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally *et al.*, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 83–es.

[16] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[18] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 829–842.

[19] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "Coda: Enabling co-location of computation and data for multiple gpu systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 1–23, 2018.

[20] K. Punniyamurthy and A. Gerstlauer, "Tafe: Thread address footprint estimation for capturing data/thread locality in gpu systems," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 17–29.

[21] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-centric data and threadblock management for massive gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1022–1036.

[22] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–27, 2020.

[23] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 583–595.

[24] L.-J. Chen, H.-Y. Cheng, P.-H. Wang, and C.-L. Yang, "Improving gpgpu performance via cache locality aware thread block scheduling," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 127–131, 2017.

[25] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 260–271.

[26] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 297–311, 2017.

[27] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 119–130.

[28] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 600–611.

[29] E. Rubin, E. Levy, A. Barak, and T. Ben-Nun, "Maps: Optimizing massively parallel applications using device-level memory abstraction," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–22, 2014.

[30] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[31] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[32] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–9.

[33] NVIDIA, " CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit, 2007.

[34] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, " Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[35] L.-N. Pouchet, " Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[36] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: The missing piece of the multi-gpu puzzle," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[37] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 128–131, 2019.

[38] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021 (To appear).

[39] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[40] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, "Tida: High-level programming abstractions for data locality management," in *International Conference on High Performance Computing*. Springer, 2016, pp. 116–135.

[41] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2009, pp. 172–187.

[42] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 23–33.

[43] W. Ma and G. Agrawal, "An integer programming framework for optimizing shared memory use on gpus," in *2010 International Conference on High Performance Computing*. IEEE, 2010, pp. 1–10.

[44] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and precise array access analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 1, pp. 65–109, 2002.

[45] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 86–97, 2010.

[46] J. Shirako, A. Hayashi, and V. Sarkar, "Optimized two-level parallelization for gpu accelerators using the polyhedral model," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 22–33.

[47] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 225–234.