Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation

Citation for published version (APA):
Papadimitriou, M., Fumero Alfonso, J., Stratikopoulos, A., & Kotselidis, C-E. (Accepted/In press). *Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation*. 57-70. Paper presented at The 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21).

Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation

Michail Papadimitriou
The University of Manchester
United Kingdom
michail.papadimitriou@manchester.ac.uk

Juan Fumero
The University of Manchester
United Kingdom
juan.fumero@manchester.ac.uk

Athanasios Stratikopoulos
The University of Manchester
United Kingdom
athanasios.stratikopoulos@manchester.ac.uk

Christos Kotselidis
The University of Manchester
United Kingdom
christos.kotselidis@manchester.ac.uk

Abstract

Although Graphics Processing Units (GPUs) have become pervasive for data-parallel workloads, the efficient exploitation of their tiered memory hierarchy requires explicit programming. The efficient utilization of different GPU memory tiers can yield higher performance at the expense of programmability since developers must have extended knowledge of the architectural details in order to utilize them.

In this paper, we propose an alternative approach based on Just-In-Time (JIT) compilation to automatically and transparently exploit local memory allocation and data locality on GPUs. In particular, we present a set of compiler extensions that allow arbitrary Java programs to utilize local memory on GPUs without explicit programming. We prototype and evaluate our proposed solution in the context of TornadoVM against a set of benchmarks and GPU architectures, showcasing performance speedups of up to 2.5x compared to equivalent baseline implementations that do not utilize local memory or data locality. In addition, we compare our proposed solution against hand-written optimized OpenCL code to assess the upper bound of performance improvements that can be transparently achieved by JIT compilation without trading programmability. The results showcase that the proposed extensions can achieve up to 94% of the performance of the native code, highlighting the efficiency of the generated code.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**.

Keywords: GPU, JIT-Compilation, Tiered-memory

ACM Reference Format:
Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2021. Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454014

1 Introduction

Heterogeneous hardware accelerators, such as GPUs and FPGAs, have become prevalent across different computing domains for accelerating mainly highly data-parallel workloads. In particular, GPUs have gained traction for accelerating general-purpose workloads due to their fine-grain parallel architecture that integrates thousands of cores and multiple levels of the memory hierarchy. In contrast to traditional CPU programming, GPUs contain programmable memory that can be explicitly utilized by developers. Although this results in gaining full control of where data can be placed, it requires extensive architectural knowledge. The majority of programming languages used for programming GPUs (e.g., OpenCL, CUDA, OpenACC) expose to their APIs specific language constructs that developers must explicitly use in order to optimize and tune their applications to harness the underlying computing capabilities.

Recently, the trade-off between GPU programmability and performance has been an active research topic. Proposed solutions mainly revolve around polyhedral models [16, 49] or enhanced compilers for domain-specific languages, such as Lift [46] and Halide [37]. These approaches either have high compilation overhead [7], which makes them unsuitable for dynamically compiled languages, or they still require developers' intervention to exploit the memory hierarchy of GPUs through explicit parallel programming constructs [37, 46].

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04...$15.00
https://doi.org/10.1145/3453933.3454014

In this paper we propose an alternative approach for automatically exploiting the memory hierarchy of GPUs completely transparently to the users. Our approach is based on Just-In-Time (JIT) compilation and abstracts away low-level architectural intricacies from the user programs, while making its application suitable in the context of dynamically compiled languages. The proposed compiler extensions are in the form of enhancements to the Intermediate Representation (IR) and associated optimization phases, that can automatically exploit local memory allocations and data locality on GPUs. We implemented the proposed compiler extensions and optimizations in the context of TornadoVM [12, 21], an open-source framework for accelerating managed applications on heterogeneous hardware co-processors via JIT compilation of Java bytecodes to OpenCL.

The proposed compiler optimizations for exploiting and optimizing local memory have been evaluated against a set of reduction and matrix operations across three different GPU architectures. For our comparative evaluation we use two different baseline implementations: (i) the original code produced by TornadoVM that does not exploit GPU local memory, and (ii) hand-written optimized OpenCL code. The performance evaluation against the original non-optimized code produced by TornadoVM, shows that the proposed compiler extensions for exploiting local memory can achieve up to 2.5x performance increase. In addition, we showcase that our proposed extensions can achieve up to 97% of the performance of hand-written optimized OpenCL code, when compared to the optimized native code.

In detail, this paper makes the following contributions:

- It presents a JIT compilation approach for automatically exploiting local memory of GPUs.
- It extends the capabilities of compiler snippets to express local memory optimizations by introducing *compositional compiler intrinsics*, that can be parameterized and reused for different compiler optimizations.
- It evaluates the proposed technique across a variety of GPU architectures, against the functionally equivalent auto-generated unoptimized and the handwritten optimized OpenCL code. Our solution achieves performance speedup of up to 2.5x versus the original code produced by TornadoVM, while reaching up to 94% of the performance of the manually optimized code.

2 Background

This section gives an overview of the memory hierarchy of GPUs using the OpenCL [34] memory model. In addition, it discusses current techniques for exploiting it, while highlighting their advantages and disadvantages.

2.1 Overview of the OpenCL Memory Model

OpenCL provides cross-platform portability for parallel code running on heterogeneous hardware, such as CPUs, FPGAs, and, most commonly, GPUs. The parallel code that is offloaded on the GPU corresponds to a `kernel`, which is submitted to the device for execution. This code is executed by the GPU Compute Units (CUs). Each Compute Unit has several Processing Elements (PEs) which are considered as virtual scalar processors. These PEs can execute on multiple threads known as `work-items`, which are grouped into `work-groups`. Furthermore, each CU can execute a number of work-groups.

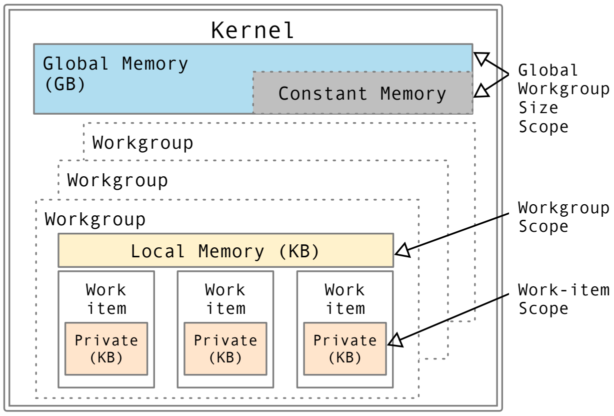


Figure 1. Overview of the OpenCL memory model.

Additionally, OpenCL provides its own memory model that consists of four memory tiers (Figure 1): Global memory provides a space that allows read/write privilege to all work-items deployed by the OpenCL device driver. Global memory encapsulates the constant memory tier, which can be allocated for read-only accesses across all work-items. The third memory tier is local memory, which can be accessed (read/write) by all work-items within the same work-group with the use of synchronization barriers [19]. Finally, the last memory tier is private memory which belongs exclusively to one work-item for storing data to a number of registers.

The GPU memory hierarchy is similar to the memory hierarchy of conventional CPUs. The global memory (in the range of GBs) corresponds to the main memory, whereas local memory (up to hundreds of KBs) corresponds to the L2 cache as it is shared among multiple work-items. Finally, private memory (up to tens of KBs) is exclusive for each work-item, and it is therefore semantically equivalent to the L1 cache of the CPU. However, unlike CPUs, which have hardware support for cache coherency, GPUs require communication barriers for coherency. In addition, the access latency between the different memory tiers of GPUs can vary in a range from ~40 to ~450 cycles for local and global memory, respectively [51]. Thus, it is essential for developers to manually explore for an optimal point in the GPU memory hierarchy for storing data, in order to achieve high performance when processing large volumes of data.

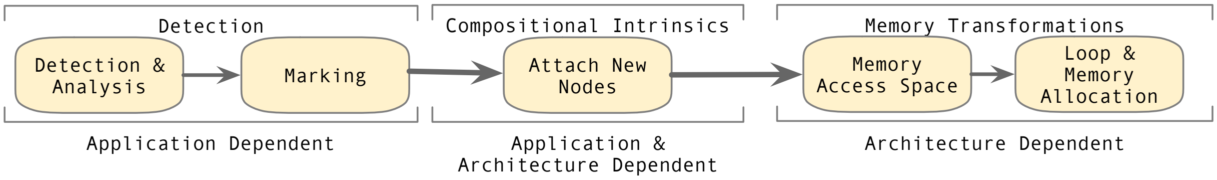


Figure 2. Overview of the proposed JIT compilation flow for automatically exploiting the GPU memory hierarchy.

2.2 Data Locality & Loop Transformations

Data locality is crucial for performance on the vast majority of applications executed on both homogenous and heterogeneous computing systems. Modern optimizing compilers targeting CPUs improve the spatial and temporal locality of coherent caches by employing optimizations, such as loop transformations. These transformations attempt to alleviate cache misses, while reducing any bank conflicts and TLB misses [32]. Therefore, optimizing compilers apply a number of loop transformations, such as loop unrolling, loop tilling, and loop un-switching [1, 2, 33].

Loop transformations have been also studied on GPUs [48]. For instance, loop tiling has been used to improve data locality and load balancing among the parallel threads on GPUs; since data split in smaller batches (tiles) can be accessed more efficiently, thereby improving the spatial locality. Excluding the programmability effort for manually achieving loop tiling, a prime challenge for compilers is the decision for the optimal tile size. This decision must be adaptive to the memory characteristics, such as the size of local memory. Thus, the decision for the optimal tile size has high complexity and is often taken based on heuristics. Polyhedral compilation [8, 16, 24, 49] is currently the state-of-the-art approach to automatically apply loop tiling for code targeting GPUs. This approach can yield high performance and often performs comparably to manually optimized libraries [4]. Nonetheless, polyhedral compilers are more suitable for ahead-of-time compilation due to the overhead in the analysis phase and the code generation [41], compared to other traditional compilers (e.g. Java HotSpot C1/C2).

2.3 Enabling GPU Tier-Memory via JIT compilation

As mentioned in the previous subsection, and will be further discussed in Section 5, current polyhedral approaches for exploiting the GPU memory hierarchy at compile time are not viable for dynamically compiled languages, due to their increased overhead in the analysis and code generation phases. Hence, current solutions for exploiting and optimizing local memory of GPUs typically expose low-level programming constructs [40] to the API. This way, the responsibility is passed to developers who must have advanced architectural knowledge to utilize the memory tiers efficiently and safely. In this work, we present a technique that allows JIT compilers to use local memory and perform loop tiling, transparently to the developers.

3 GPU Memory-Aware JIT Compilation

This section presents our main contributions towards automatically exploiting and optimizing the memory hierarchy on GPUs via JIT compilation.

3.1 Overview

Figure 2 presents an overview of the JIT compilation process for exploiting local memory. The proposed approach includes three different phases: *detection*, *compositional intrinsics*, and *memory transformations*. All phases are applied to the common IR of the TornadoVM JIT compiler (which is a superset of the Graal IR [18]). The TornadoVM IR uses the sea-of-nodes [13] common representation, which encompasses both the control-flow and data-flow nodes. This representation allows the compilation and optimization of Java bytecodes to OpenCL by performing IR transformations.

The ***detection*** phase scans the IR to locate specific nodes, such as accesses to/from arrays through read and write nodes, as well as the induction variables. To use local memory, nodes that are commonly used to read and write from/to memory are first located (by default, the JIT compiler assumes all accesses target GPU's global memory). These array accesses are detected via indexed read and write nodes in the IR. The detection phase is crucial for the compilation process as: a) it provides the exact place in the IR to read/write from/to local, instead of global memory, and b) it analyzes all nodes accessible from the indexed read/write nodes, such as the induction variables and the parameters of the compiled method. This information is accounted during the detection phase to introduce and attach a new node. The newly introduced node encloses the read and write nodes and it is used by the next phases to perform aggressive optimizations regarding GPUs' local memory (Sections 3.3) and loop tiling (Section 3.4).

The ***compositional intrinsics*** phase adds to the IR the nodes needed for performing memory allocation, and prepares the IR for code generation. In a nutshell, this phase starts by specializing the IR for GPU architectures based on the new nodes that were trailed from the detection phase. Although the previous phase was only application dependent, from this stage and onwards application and architecture dependent optimizations are being applied to the IR. During this process, the high-level IR is lowered into a more concrete lower IR (known as the lowering process) which has a closer mapping to the underlying target architecture. Since this process involves the introduction of a number of

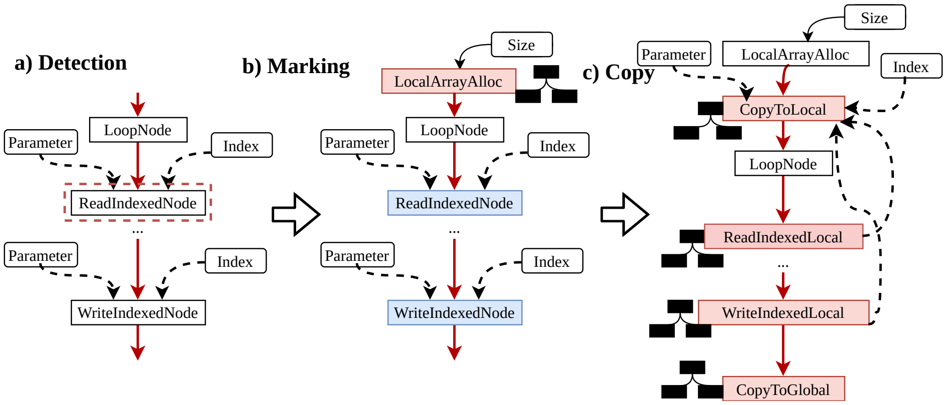


Figure 3. IR transformations for the compiler intrinsics of local memory allocation, and data copies.

new IR nodes, we opted for creating and utilizing a set of parameterized compiler intrinsics that can be composed at run time to form larger graphs. These compiler intrinsics are in the form of *snippets* [42] and they are essentially methods, completely written in Java, that represent low-level operations that are being attached to the IR at runtime. Section 3.2 explains in detail, all the compiler intrinsics introduced for automatically utilizing local memory.

Finally, the ***memory transformations*** phase is an architecture dependent optimization process. In this phase, the JIT compiler processes the new nodes introduced during lowering, and completes the IR with the correct information to access local memory. This low-level information includes the base addresses and the offset arithmetic nodes. In summary, this phase introduces new IR operations for: 1) copying data from global to local memory, 2) materializing the indices to read/write from/to local memory, and 3) copying the final data from local to global memory upon finishing executing a kernel. In addition, this phase invokes the OpenCL API for obtaining device specific information to optimize local memory sizes, based on the number of work-items deployed and the available local memory.

3.2 Compositional Compiler Intrinsics

Compiler intrinsics are low-level code segments that are typically expressed in low-level programming languages, such as assembly or C. They represent optimized code for common operations, such as the use of vector operations or memory allocation. The JIT compilers of Graal and MaxineVM [31, 50] introduced the concept of *compiler snippets* as a high-level representation of low-level operations [42]. With snippets, low-level operations are implemented in a high-level programming language (Java) instead of the assembly code. Since the aforementioned JIT compilers are also implemented in Java, they do not need to cross language boundaries to implement their intrinsics and, hence, their code can be further optimized by applying common compiler optimizations (e.g., loop unrolling, constant propagation, etc.).

Fumero *et al.* [20] extended the use of compiler snippets to express efficient parallel skeletons for GPUs in TornadoVM. In this paper, we extend the capabilities of compiler snippets to express local memory optimizations by introducing *compositional compiler intrinsics*, that can be parameterized and reused for different compiler optimizations. With this approach, we can further increase the performance of input applications by automatically exploiting local memory.

We implemented a set of parameterized compiler intrinsics that allow us to gradually lower the IR and generate efficient GPU code that makes use of local memory. These compiler intrinsics are involved in two different compilation phases: the *compositional intrinsics* phase, in which we insert the actual compiler intrinsics into the compiled graph (IR), and the *memory transformations* phase, in which the IR is optimized after inlining the intrinsics into the graph. This approach offers a degree of flexibility to the compiler to apply a number of optimizations, as well as to combine intrinsics to express multiple optimizations. In detail, the following intrinsics are introduced:

Local Memory Allocation: This intrinsic modifies the IR to emit code for allocating arrays in local memory. Input and output variables that have been detected in the first phase, are marked as candidates for using local memory. In this case, this compiler intrinsic introduces the logic to declare and instantiate arrays in local memory. By design, snippets do not support dynamic memory allocation, and consequently, the *Local Memory Allocation* intrinsic does not either. Therefore, array lengths have to be statically set. To address this limitation, we provide the lengths of the arrays to be stored in local memory as a parameter node that can be dynamically changed and updated in the *memory transformations* phase. The actual size depends on the amount of local memory available on the target device and the number of threads to be deployed. In this way, multiple combinations of local memory sizes can be generated during runtime. Figure 3 illustrates the use of this compiler intrinsic in our JIT Compiler. The left-hand side of Figure 3 shows the IR that represents an indexed read and an indexed

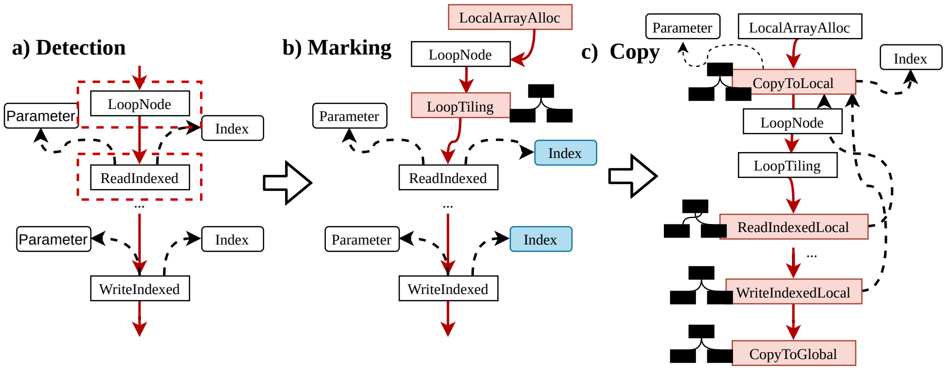


Figure 4. IR transformations for the compiler intrinsics of loop tiling, local memory allocation, and data copies.

write from/to an array inside a loop. The graph is read as follows: the control flow nodes are connected with red arrows, while the data-flow nodes are connected with black dashed arrows. In addition, the introduction of a compiler intrinsic is represented by a red node, while a blue node represents a node needed to perform an optimization. In this phase, the JIT compiler runs the detection phase, looking for reads and writes that are enclosed in loops. Upon the detection of the `ReadIndexedNode`/`WriteIndexedNode` nodes (Figure 3(a)), the compiler marks them as candidates to use local memory and introduces a set of new nodes (i.e., `LocalArrayAlloc`, `Size`) in the IR (Figure 3(b)).

Copy To Local Memory/Copy To Global Memory: These compiler intrinsics introduce a copy from global to local memory, and vice versa. These memory copies are presented in Figure 3(c) by two new IR intrinsics `CopyToLocal` and `CopyToGlobal`. Both intrinsics are performed during the *memory transformations* phase and accept as inputs the local array nodes and the corresponding indices from global and local memory.

Load/Store Operations in Local Memory: This pair of intrinsics performs loads and stores operations from arrays that reside in local memory to private memory, and vice versa. Figure 3(c) illustrates these operations with two new IR intrinsics `ReadIndexedLocal` and `WriteIndexedLocal` that represent the load and store operations, respectively. This pair of compiler intrinsics enables our JIT compiler to access the local memory address space, as opposed to the TornadoVM IR indices (`ReadIndexed` and `WriteIndexed` in Figure 3(b)) that do not support this functionality.

Reductions with Local Memory: This intrinsic improves the reduction operations proposed by Fumero *et al.* [20], by adding local memory support. By using the same technique as described for the two previous compiler intrinsics, we utilize the GPU local memory so as to increase the performance of reduction operations on GPUs. Section 3.3 explains all the IR transformations involved to generate efficient GPU reductions using local memory via our compiler intrinsics.

Parameterized Loop Tiling for Local Memory: We also introduced a set of compiler intrinsics that can be combined with common loop optimizations, such as loop tiling and loop unrolling. Although these loop optimizations are orthogonal to the use of local memory, they can facilitate the use of local memory. To do so, we introduced a compiler intrinsic in the JIT compiler to perform loop tiling. This intrinsic receives, as parameters, all arrays stored in local memory and all loop indices that access local memory. Through the parameterized architectural design of the compiler intrinsics, we can further combine this optimization with loop unrolling. Figure 4 illustrates an example of this compiler intrinsic that combines the local memory allocation with loop tiling. Figure 4(a) shows the detection phase with three primary nodes: a loop node and two indexed read and write nodes. During the detection phase the loop node is selected as a candidate node for loop-tiling. The second graph shows the expansion of the IR through the introduction of the compiler intrinsic for loop-tiling. This new set of nodes in the IR enables a new marking phase to apply local memory and loop tiling (Figure 4(b)). Figure 4(c) shows the new IR after applying local memory allocation, loop tiling, and the copies from global to local memory (and vice versa once the loop tiling optimization is performed).

For the rest of this section, we use two different use cases to showcase how compositional compiler intrinsics are introduced in the IR, and how they are optimized to efficiently utilize the GPU memory hierarchy. Note that, although we demonstrate our approach in the context of the TornadoVM JIT compiler, the proposed technique can be used by other compilation frameworks that provide similar features, such as LLVM and GCC.

3.3 Exploiting Local Memory: Parallel Reductions

The first use-case that we utilize to showcase the proposed technique regards the reduction operations, which are defined as the accumulation of input values from a vector into a single scalar value. Reductions are one of the basic primitives for many parallel programming frameworks, such as

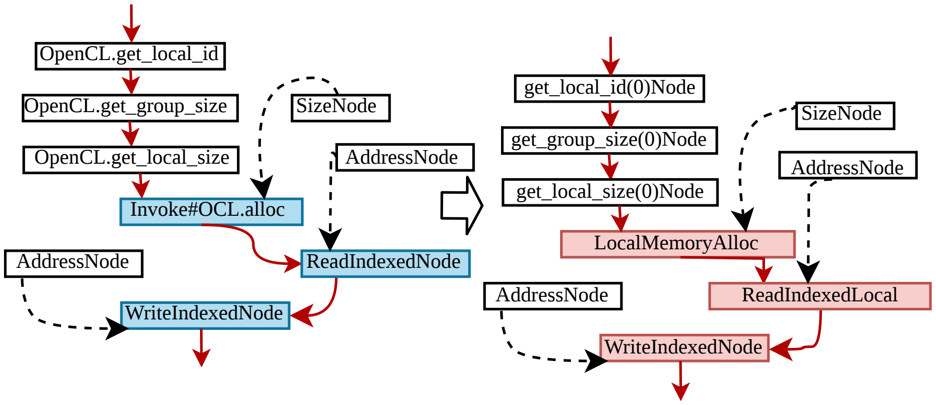


Figure 5. Node replacements during the lowering phase for the reduction compiler intrinsic.

Google Map/Reduce [15], Apache Spark [53], Flink [9], and common libraries such as Thrust [6]. Therefore, optimizing parallel reductions has been a well studied topic, especially with regards to memory optimizations such as local memory [10, 14].

To perform high performance reductions on GPUs, TornadoVM currently makes use of compiler intrinsics [42] to express parallel skeletons [20]. TornadoVM already solves the problem of seamlessly expressing parallel reductions in the compiler, albeit without exploiting data locality and GPU local memory. As follows, we describe each compiler phase of the proposed technique for adding local memory support to reduction operations.

Detection. To express reductions in TornadoVM, developers use the *@Reduce* annotation. Upon adding the annotation, the TornadoVM JIT compiler detects the reduction pattern which is subsequently used by our solution to add local memory support.

In a nutshell, TornadoVM targets only the global memory space by automatically dividing the iteration space in smaller chunks (one chunk per work-group), and it performs a full reduction within each chunk.

Lowering. TornadoVM implements parallel reductions with the use of intrinsics (further information can be found in [20]). Listing 1 exemplifies the compiler intrinsic (snippet) that is used to perform a reduction operation. As shown, the compiler intrinsic is also written in Java and during compilation its generated IR is appended to the rest of the compiled method's IR graph. Consequently, the merged IR can be re-optimized iteratively; a key advantage in comparison to intrinsics written in low-level languages which are treated as native functions from the compiler.

We augmented the existing intrinsic to add support for local memory by adding the statements in gray color (Listing 1). To achieve this, we implemented additional compiler intrinsics to express local memory regions in a high-level

```
1  @CompilerIntrinsinc
2  void reductionIntrinsic(float[] input,
3    float[] output){
4    int idx = OpenCL.get_local_id(0);
5    int lgs = OpenCL.get_local_size(0);
6    int gID = OpenCL.get_group_id(0);
7    float[] local = OCL.alloc(SIZE, float.class);
8    local[idx] = input[OpenCL.get_global_id(0)];
9    for(int i = (lgs/2); i > 0; i/=2) {
10     OpenCL.localBarrier();
11     if (idx < i) local[idx] *= local[idx + i];
12   }
13   if (idx == 0) output[gID] = local[0];
14 }
```

Listing 1. Code of a compiler intrinsic in our JIT compiler to utilize the GPU's local memory for reductions.

manner. In this case, we explicitly use a local memory region by allocating the corresponding arrays in the generated OpenCL source code, instead of defining a parameter to the generated OpenCL kernel with a local memory region. Line 6 shows the allocation of the `local` array in local memory. Note that the allocation is performed via an invocation to the static method `OCL.alloc`, in which we pass the size and the type of the array. Consequently, line 7 copies data from global memory to local memory. Then, the actual reduction is computed using local memory (line 10). Finally, line 12 performs the final copy from local to global memory. These intrinsics are lowered by the JIT compiler to generate OpenCL C code that corresponds to the high level Java code. By using this strategy of computing with local memory, the execution flow from global memory is transformed to local memory.

During the lowering phase, the IR generated by the compiler intrinsic includes new nodes associated with allocating, indexing, and storing data to the local memory region. Then, the new nodes are inlined to the IR graph of the compiled

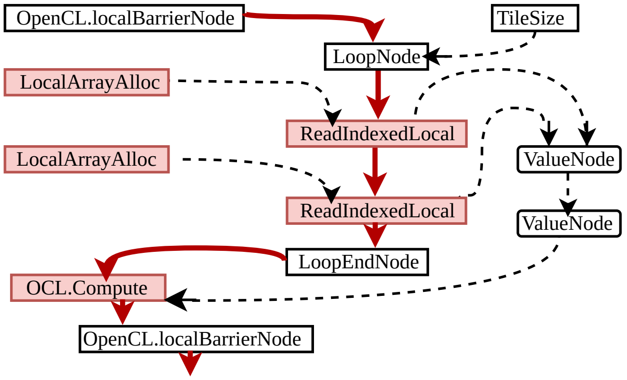


Figure 6. IR nodes from the compiler intrinsic in Listing 2.

```
1  @CompilerIntrinsinc
2  void tile(float sum, float[] arrA, float[] arrB,
3            int size, ValueNode operator,
4            ValueNode reduceOperator) {
5    OpenCL.localBarrier();
6    for (int x = 0; x < size; x++) {
7      sum = OCL.compute(arrA[x], arrB[x],
8                operator, reductionOperator);
9    OpenCL.localBarrier();
10   }
11 }
```

Listing 2. Example of a compositional compiler intrinsic for processing loop tiling using local memory.

method. Figure 5 depicts the IR transformations upon replacing the IR nodes introduced by the intrinsic in Listing 1 with the corresponding lowered IR nodes (via substitution) for local memory allocation. Similar to Figure 3, control-flow nodes are connected with red arrows, while data-flow nodes are connected using black dashed arrows. The left graph in Figure 5 represents the IR when the code for the reduction intrinsic is built. This graph includes the `Invoke#OCL.alloc` node that represents an array allocation using local memory. This node contains information about the size, that is used as a data-flow node, allowing us to dynamically change the size. Therefore, the same compiler intrinsic can generate parameterizable code for various local memory sizes. The right graph shows the IR graph after applying the substitution to allocate local memory.

Memory Transformations. A challenge in this phase is that the upfront decision for the allocated size of local memory has to be taken in accordance with the deployed GPU threads (work-items). However the number of deployed threads is determined at runtime and depends on the input data size of the application. To tackle this challenge, we attach the sizes of the local arrays as a data-flow node (`SizeNode`) in the IR, as illustrated in Figure 5. In this case, if the same reduction is executed, during runtime, with a different input size, the generated code will be dynamically adapted by changing only the size node that is attached to the `LocalMemoryAlloc` node in the IR.

3.4 Data Locality for Local Memory using MxM

The second use-case that we use to express the efficacy of the proposed technique is the $O(N^3)$ matrix multiplication operation. This code has three nested loops that can be parallelized via TornadoVM with the employment of the *@Parallel* annotation. This section explains all the phases in the JIT compilation flow that facilitate the utilization of data locality in the local memory.

Detection. The detection phase of our JIT compiler traverses the IR graph and seeks for the `ReadIndexed` and `WriteIndexed` nodes, which represent the memory accesses to the global memory. Figure 4(a) illustrates this process in which all the derived information about the induction variables and the parameters of the method contributes to the addition of two new nodes that apply two compiler intrinsics; one for local memory allocation and a second for loop tiling at the innermost loop.

Lowering. Figure 4(b) presents the marking of the two nodes that were added in the previous phase (`LocalArrayAlloc` and `LoopTiling`). During the lowering phase, the IR nodes are replaced by the respective compiler intrinsics. As the local memory allocation intrinsic was discussed in Section 3.2, we describe here the application of the loop tiling intrinsic. Listing 2 shows the code that implements the compiler intrinsic in our JIT compiler for loop tiling. This intrinsic accepts as inputs a set of arrays, the size for the loop tiling and the operators to be applied inside the loop tiling. Line 6 shows the new loop to perform the tiling and line 7 shows a method invocation that introduces the compute logic inside this new loop. Note also that two OpenCL local barriers are required to guarantee consistency. The first barrier in line 5 is used prior to loop tiling to ensure that the data have been copied to the allocated space in the local memory, whereas the second barrier (line 9) synchronizes the processing of the tile across all work-items before the final copy to the global memory. Note that developers do not need to worry about maintaining memory consistency when using local memory, since the barriers are automatically inserted by the JIT compiler. Figure 6 shows the IR representation for this compiler intrinsic. The new loop is introduced as a control flow node (`LoopNode`) right after the node of the OpenCL local barrier. The loop body is represented by a compiler intrinsic called `OCL.Compute`. This intrinsic acts as a placeholder for inserting the IR nodes that represent the core computation within the loop tiling, which, in the case of matrix multiplication, it corresponds to a multiplication, followed by a sum. In turn, all these new nodes will be replaced during the memory transformations phase.

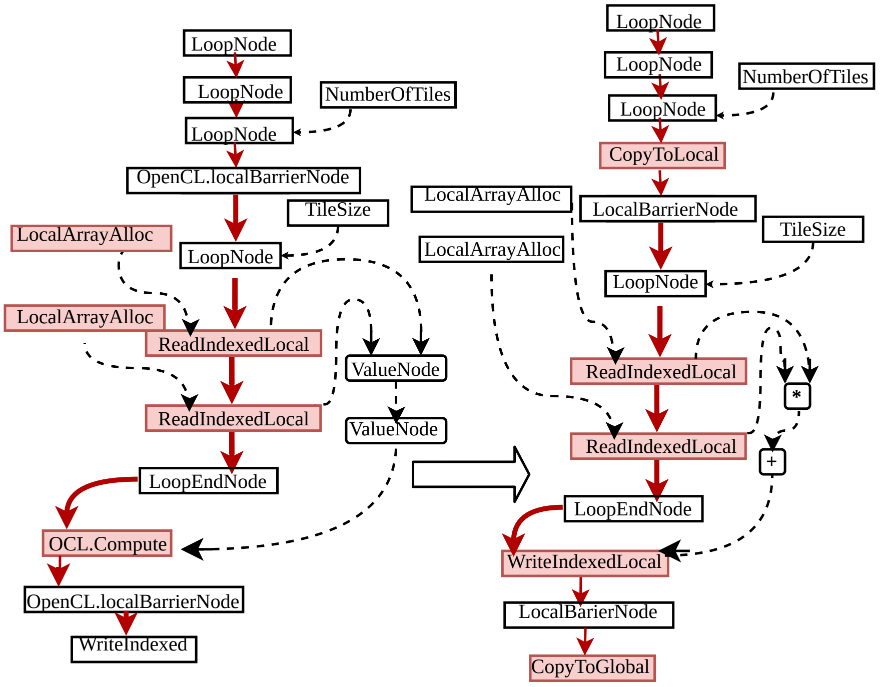


Figure 7. IR node replacements during the memory transformation phase for the Matrix Multiplication application.

Memory Transformations. Figure 7 illustrates the transition of the IR from lowering (left graph) to the final memory transformations phase (right graph). In the last phase before the OpenCL C code generation, a set of new compiler intrinsics (e.g., `CopyToLocal`, `CopyToGlobal`) is introduced to use local memory. The `WriteIndexedLocal` intrinsic of the right graph in Figure 7 is used to store the final result from the `sum` variable (Listing 2 - line 7). This phase has been previously discussed with regard to the local memory allocation in Section 3.3.

Regarding the loop tiling optimization, the left graph in Figure 7 shows the IR of the loop tiling compiler intrinsic (Figure 6). During the *memory transformations* phase all the IR nodes of the compiler intrinsic are lowered to OpenCL instructions. In particular, this phase inlines the call of the `OCL.Compute` method that was introduced in the previous phase into a set of nodes that performs the computation of the method. In this case, the call inlines all nodes involved in the matrix multiplication operation within the loop tiling (see right graph in Figure 7).

As the loop tiling compiler intrinsic is applied to the innermost loop, three more nodes (`LoopNode`) are illustrated in Figure 7 representing the three outermost loops. Therefore, the lowering process of loop tiling starts by first traversing the IR graph from the innermost loop, and then replacing its loop bound with a `TileSize` node, and the bounds of the third innermost loop with a `NumberOfTiles` node. The two outermost loops remain the same as they represent the sizes of parallel dimensions. To decide the tile size during JIT compilation, the OpenCL driver is invoked to provide the maximum number of the available work-items which is device-specific. Similarly, the number of deployed threads (`GlobalWorkItems`) is obtained from the OpenCL driver as it matches the input data size of the application. This information is used to calculate the number of total tiles.

Finally, due to our parameterizable compiler intrinsics, existing compiler intrinsics can be combined with more aggressive optimizations, such as loop unrolling and partial escape analysis.

4 Evaluation

This section presents the performance evaluation of the proposed technique against two baseline implementations: (i) the original code produced by TornadoVM[1] that does not exploit GPU local memory, and (ii) hand-written optimized OpenCL code. The OpenCL baseline implementation includes the same set of optimizations as our extended JIT compiler. Table 1 presents the hardware specifications of the three GPU devices used in our testbed. The system runs CentOS 7.4 with Linux kernel 3.10, and for all experiments we use the OpenJDK JVM 1.8 (u242) 64-Bit with 16GB of Java heap memory. In order to ensure that the JVM has been warmed up, we execute 100 iterations per benchmark, and we report the geometric mean.

[1] The exact commit point is: 81c70437800c252899a56e78ddbe80697f273973.

Table 1. Device and driver specification for the experimental setup.

Device	Vendor	Work-Items	Global	Local	Driver
GFX900	AMD	1024x1024x1024	8GB	64KiB	2766.4
GeForce 1650	Nvidia	1024x1024x64	4GB	48KiB	435.21
HD Graphics	Intel	256x256x256	25GB	64KiB	19.43.14583

Table 2. The list of benchmarks used in the evaluation.

Benchmark	Input Sizes	Method/Kernel LOC			Opts	
		Java	Gen	OpenCL	Lc	Tile
Reduction (Min)	2^8 to 2^{24}	5	40	19	Y	N
Reduction (Add)	2^8 to 2^{24}	5	40	19	Y	N
Reduction (Mul)	2^8 to 2^{24}	5	40	19	Y	N
Transpose Matrix	2^8 to 2^{24}	6	77	14	Y	N
Matrix Multiplication	2^5x2^5 to 2^{12}x2^{12}	11	63	25	Y	Y
Matrix Vector Multiplication	2^6x2^3 to 2^{16}x2^8	9	55	20	Y	Y

4.1 Benchmarks

We evaluate our technique against three reduction operations (*Minimum*, *Addition*, and *Multiplication*), and three matrix operations (*Matrix Multiplication*, *Matrix Transpose*, and *Matrix Vector Multiplication*). Table 2 presents the various parameters used for each benchmark including the input data size, the lines of code (LOC), and the combination of optimizations applied per benchmark; namely local memory usage (Lc) and loop tiling (Tile). The evaluated benchmarks have been implemented in Java, for execution with TornadoVM, and in OpenCL C for comparisons against hand-written optimized native code. The third column (Java) of Table 2 shows the LOC for the TornadoVM Java implementations, while the fourth column (Gen) shows the LOC of the auto-generated GPU code. Finally, the fifth column (OpenCL) shows the LOC of the manually written OpenCL C codes. We present the LOC of the implementations in order to provide an insight of the complexity of the developed code with respect to utilizing the GPU memory hierarchy. Note that the OpenCL code generation in TornadoVM (Gen) derives from SSA (Static Single Assignment) representation (in which each operation is assigned exactly once). Therefore, more lines of code are generated. Regarding optimizations, all reductions exploit local memory as explained in Section 3.3, whereas the matrix operations exploit the different combinations (Lc, Tile), as discussed in Section 3.4.

4.2 Performance Comparison vs. TornadoVM

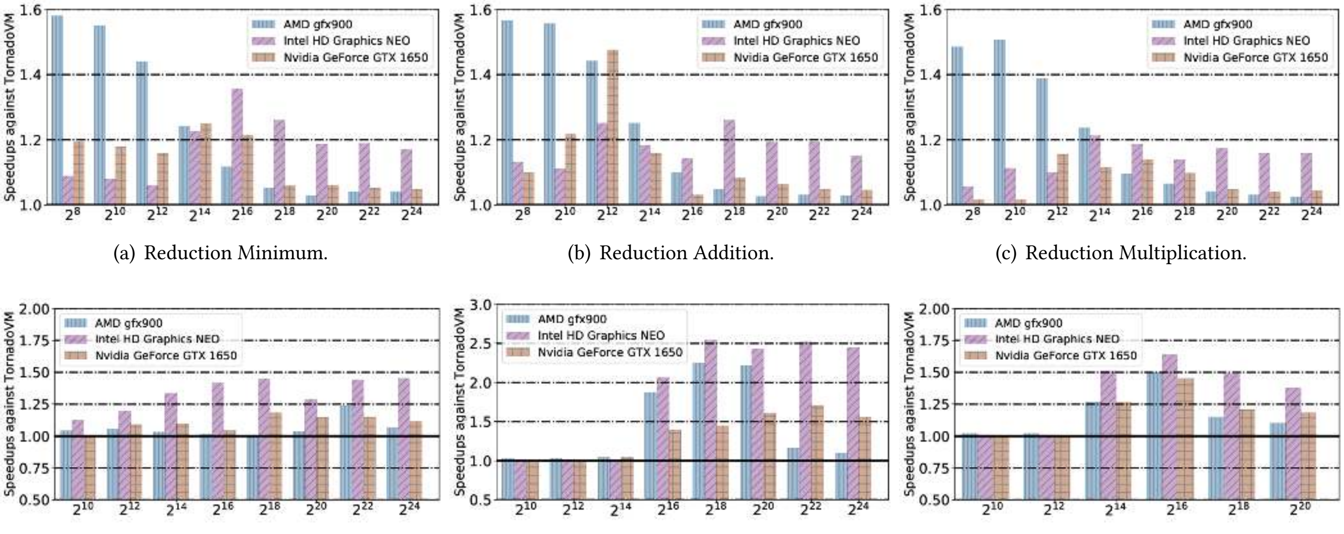
Figure 8 presents the performance speedup achieved by the proposed compiler optimizations, against TornadoVM which does not support local memory. For both figures, the x-axis shows the input size for each benchmark, while the y-axis shows the speedup against TornadoVM.

In general, our approach outperforms TornadoVM by up to 2.5x and 1.6x for matrix and reduction operations, respectively. Additionally, all benchmarks exhibit performance speedups across all data sizes. For Intel and NVidia GPUs, the reported times include only the kernel execution on the GPUs. On the contrary, for the AMD GPU the reported times include also data transfers. This is due to a limitation of the AMD OpenCL driver which can only report kernel execution and data transfer times combined. For this reason we also separate the discussion regarding performance between the different GPUs.

AMD GPU Performance. As shown in Figure 8, our compiler optimizations yield performance speedups ranging from 1.02x to 1.58x on the AMD GPU. Regarding all reduction operations (Figure 8(a-c)), we observe that the execution for small input data sizes yields higher performance compared to larger input sizes when utilizing local memory (up to 1.58x at 2^8 data elements in Figure 8(a)). Since the reported times of the AMD GPU include also data transfers, the observed speedups degrade as the input data sizes increase due to the costly data transfers. Nevertheless, these overheads do not result in slowdowns. Regarding matrix operations (Figure 8(d-f)), the execution on the AMD GPU obtains a maximum performance of 2.3x for matrix multiplication and 1.23x for matrix transpose, following similar trends with the reduction operations.

Nvidia and Intel GPU Performance. As shown in Figure 8, the execution with local memory on Intel HD Graphics (second bars) performs up to 35% faster than the baseline configuration (2^{16} data elements in Figure 8(a)). Regarding the execution on the Nvidia GPU (third bars), performance improvements of up to 48% are observed (2^{12} data elements in Figure 8(b)). As the data sizes increase, the relative performance speedups of the proposed optimizations decrease. This is attributed to the additional global barrier that had to be placed into the generated code before the final read from local to global memory. As the number of threads increases and surpasses the amount of physical threads that can run in parallel on the device, the overhead of the barrier also increases. We plan to address the barrier overheads by applying node hoisting in future work.

Concerning matrix operations (Figure 8(d-f)), the largest speedup (up to 2.5x) is observed when running on the Intel HD Graphics (2^{18} data elements in Figure 8(e)). In general, the observed speedups for matrix operations are higher than those in reduction operations, mainly due to the combination of the applied optimizations (i.e., loop tiling and local memory). Finally, as shown in Figure 8(e-f), for small data sizes we observe no performance increases. This is attributed to the loop unrolling optimization taking place at the early



(a) Reduction Minimum.

(b) Reduction Addition.

(c) Reduction Multiplication.

(d) Matrix Transpose.

(e) Matrix Multiplication.

(f) Matrix Vector Multiplication.

Figure 8. Performance comparison against vanilla TornadoVM (the higher, the better). The x-axis represents the input size for each benchmark, while the y-axis shows the performance speedup against TornadoVM.

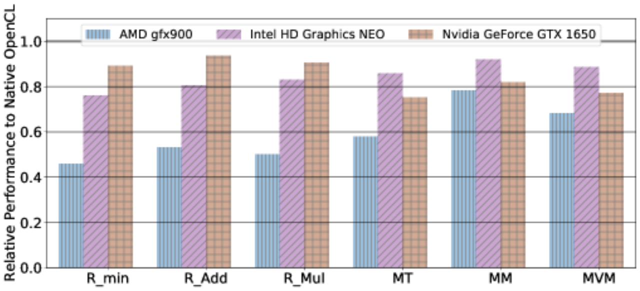


Figure 9. Relative performance of the code generated by our extended JIT compiler against hand-written optimized OpenCL implementation (the higher, the better).

Table 3. Compilation times per phase.

	Time (ms)			
Benchmark	**TornadoVM**	**Nvidia Driver**	**Intel Driver**	**AMD Driver**
Reduction Add	64.59	47.04	224.38	18.54
Reduction Mul	73.23	54.60	251.16	19.64
Reduction Min	81.38	57.70	258.61	18.85
Matrix Transpose	55.43	43.20	227.73	17.42
Matrix Mul.	62.21	48.10	250.68	21.32
Matrix-vector Mul.	61.31	52.40	254.68	19.32
GeoMean	**65.81**	**50.39**	**239.06**	**19.16**

stage of the optimizations which consequently negates the local memory optimizations proposed in this paper. Nevertheless, it is possible to apply the proposed optimizations on unrolled loops in future work.

4.3 Performance Comparison vs. Hand-Written OpenCL

Figure 9 shows the relative performance of the code generated by the JIT compiler against the functionally equivalent optimized (using local memory and loop tiling) OpenCL code. Similarly to the previous experiments, the times reported on the AMD GPU include both kernel and data transfer times in contrary to Intel and Nvidia GPUs that report only kernel times.

As shown in Figure 9, the performance of the JIT-compiled code compared to native OpenCL C implementations for reductions, reaches up to 53% on the AMD GPU, up to 83% on the Intel HD GPU, and up to 94% on the Nvidia GPU. Regarding matrix operations (Figure 9), the JIT-compiled code performs up to 78% on the AMD GPU, up to 92% on the Intel HD GPU, and up to 82% on the Nvidia GPU, compared to the native OpenCL C code. As expected, the results shown in Figure 9 demonstrate that the performance of the JIT-compiled code does not match that of the optimized OpenCL C native code. However, the auto-generated code performs competitively especially after considering the fact that no user intervention for performance tuning is required.

4.4 Compilation overhead

Table 3 presents the time spent for JIT compilation separated into two categories: TornadoVM and driver compilation times. The TornadoVM compilation time is the time taken to JIT-compile the Java bytecodes to OpenCL code, while the driver compilation times are the reported times of the device drivers for compiling the OpenCL code to machine code.

In order to better understand the JIT-compilation overheads, we investigated the *Matrix Multiplication* benchmark, since it combines both the local memory allocation and loop tiling. The JIT-compilation of that benchmark exhibits up to 63.7% of additional compilation time compared to the original TornadoVM JIT compiler. From that additional compilation time, the newly introduced optimization phases account for up to 25%. The rest of the overhead is distributed amongst the rest of compilation phases and they are attributed to the increased size of the IR graph. The addition of local memory and loop tiling awareness to the IR graph results up to 50% additional nodes that are processed by subsequent optimization. The occurrence of extra nodes that are processed by the consequent optimization phases is translated to approximately 35% increase of compilation time. In addition, the percentage of compilation time in the total execution time is less than 5% and, as in any other optimizing JIT-compilers, this overhead is encountered only once during execution (the initial compilation).

4.5 Automatically Exploiting Private Memory

Similarly to local memory, we also introduce private memory array allocation on GPUs for arrays that are allocated and used within the scope of each work-item. All Java objects (including arrays) are allocated on the Java heap. However, if the Java objects do not escape a certain scope (e.g., a method scope), modern JIT compilers might apply a compiler technique called partial escape analysis (PEA) [43]. This optimization aims to avoid the object allocation on the heap, and instead, to use the Java stack and the internal registers.

We extend this model by using the private memory array allocation for objects that do not escape the scope in which they are declared. We analyze in the IR the data access patterns, which track the usage per work-item of the declared arrays. If arrays are only used within the work-item scope, we replace the object allocation by an explicit allocation in private memory of the required sized. This means that, if the array is declared either within a sequential loop in Java (without the `@Parallel` annotation), or in a parallel loop without any data dependencies between its accesses, and the array does not escape the method's scope, then we replace the allocation from the global memory into private memory. Note that due to the limited number of physical registers, the OpenCL driver might allocate the array in global memory.

Since our extensions to TornadoVM are tightly integrated into the Graal compiler infrastructure, it is not possible to isolate this optimization and measure its effect. Additionally, by disabling PEA, arrays would be allocated in global memory; an operation which is not supported by TornadoVM. Nevertheless, we implemented a synthetic benchmark to demonstrate the effect of private memory usage in our compiler infrastructure. We run a reduction per work-item using an array allocated in private memory versus an array using global memory on an NVIDIA GPU. To avoid compiler optimizations from the NVIDIA CUDA compiler, we also disabled compiler optimizations (`cl-opt-disable`). By analyzing the PTX generated by the NVCC CUDA compiler, we found that the private arrays are allocated in constant memory, and the load operations are performed into local memory. This version performs up to 2.3x times faster than using only the global memory.

5 Related Work

This section discusses the related work regarding the exposure of GPU memory optimizations into programming languages and implementations, and optimizations techniques for memory transformations.

We classify the related work into two groups. The first group describes approaches for exposing GPU features to a wide range of high-level programming languages. The second group focuses on various memory transformations at the compiler level.

5.1 GPU Features into Programming Languages

In the context of dynamically compiled languages (e.g., Java) several frameworks [3, 21] have been proposed to exploit GPU acceleration. Aparapi [3] and TornadoVM [21] are Java-based frameworks that dynamically compile Java bytecodes to OpenCL code. Aparapi exposes specific language constructs for memory allocation (i.e., local memory) and memory synchronization (i.e., barriers) that programmers must explicitly use [3]. On the contrary, TornadoVM generates high-level bytecodes to abstract programmers from the GPU programming model. However, it does not automatically exploit fine-grain memory and does not expose low-level constructs to developers. Moreover, IBM J9 [27] is another example of a JIT compiler for GPU offloading, but it exclusively compiles Java streams to CUDA-PTX code. The only memory optimization supported by J9 is the placement of read-only data to read-only caches. Similarly, the Marawacc [23] compiler and FastR-GPU [22] only exploited global and constant memory via the Graal JIT compiler.

In addition, several parallel programming frameworks exist [11, 17, 23, 29, 38, 44, 45, 47] that enable the compilation of domain-specific languages on GPUs. Lift [26, 46] extends its existing data parallel primitive types to accommodate loop tiling (e.g., `slide,pad`) and its low-level OpenCL with local memory (e.g., `toLocal`) allocation for stencil computations. Ragan-Kelley *et al.* [37] introduced Halide, a domain-specific language (based on C++) for executing high-performance image processing code on GPUs. However, the exposure of the GPU features at the programming language increases the development cost, as the resulting performance is tightly coupled with the programmer's experience.

Our work differs from all aforementioned frameworks as we propose an approach that automatically exploits the GPU memory hierarchy, without exposing any specific language constructs to the developers.

5.2 Compiler Techniques for Memory Transformations

Verdoolaege *et al.* [49] used polyhedral models to automatically transform C code to CUDA while utilizing shared memory and loop tiling. Similarly, Bondhugula *et al.* [8] proposed PLUTO, an automatic loop nest parallelizer to exploit data locality via shared memory. In addition, Grosser *et al.* [24] have extended the polyhedral models in PLUTO with loop splitting for stencil workloads. PolyJIT of Simburger *et al.* [41] combines polyhedral optimization with multi-versioning at run time; a technique that poses significant overhead during code generation. A number of studies [4, 5, 30, 39] target loop tiling optimizations and code generation for GPUs for affine loops. Moreover, Di *et al.* [16] proposed an algorithm to improve tiling hyperplanes by using dependency analysis, while Grosser *et al.* [24] developed a polyhedral-based parametric scheme leveraging run-time exploration of partitioning parameters.

In addition, a number of non-polyhedral based approaches have been also proposed. Kim *et al.* [28] presented an approach to map tensor contractions directly to GPUs, while using shared memory by a parametric code generation strategy that utilizes a cost model for data movement. Chen *et al.* [11] extended Halide's support for new optimizations that target the memory hierarchy by introducing the concept of memory score in the compiler. Yang *et al.* [52] introduced an optimizing source to source compiler for C programs that exploits a number of memory optimizations, such as converting non-coalesced accesses, vectorization for memory access, and tiling with shared memory. Additionally, Hagedorn *et al.* [25] proposed Elevate, a new functional language to express various optimizations such as vectorization, tiling, splitting, and others.

Our work improves upon the approaches above that employ exhaustive techniques to optimize memory transformations, as it provides a trade-off between compilation time and achieved performance, making it more suitable for interpreted and dynamically compiled programming languages.

6 Conclusions

In this paper we presented an approach to efficiently exploit the memory hierarchy of GPUs from dynamically compiled languages. This is achieved by extending the capabilities of compiler snippets to express local memory optimizations by introducing compositional compiler intrinsics, that can be parameterized and reused for different JIT compiler optimizations. Our solution provides a trade-off between compilation times and achieved performance, making it suitable for JIT-compiled languages. The presented work has been prototyped in the context of TornadoVM and includes compiler extensions and optimizations to exploit GPU local memory and loop tiling. Our proposed technique has been evaluated across three GPU architectures and the results indicate that it can achieve performance speedups of up to 1.58x and 2.5x for reduce and matrix operations, respectively. We also showcased that the performance of the proposed extensions can achieve up to 94% of the performance of the manually written OpenCL code. Most importantly, the aforementioned performance increases come at no programmability costs since they are transparently applied to unmodified user programs at compile time.

In the future, we plan to apply our work to other types of computations (e.g., stencil computations) while devising further optimizations. Furthermore, we plan to expand the proposed technique for other niche accelerators (e.g., FPGAs [35, 36]).

Acknowledgments

The work presented in this paper is partially funded by grants from Intel Corporation and the European Union's Horizon 2020 E2Data 780245 and ELEGANT 957286 projects.

References

[1] 2014. Loop optimizations in Hotspot Server VM Compiler (C2). https://wiki.openjdk.java.net/pages/viewpage.action?pageId=20415918

[2] Qurrat Ul Ain, Saqib Ahmed, Abdullah Zafar, Muhammad Amir Mehmood, and Abdul Waheed. 2018. Analysis of Hotspot Methods in JVM for Best-Effort Run-Time Parallelization. In *Proceedings of the 9th International Conference on E-Education, E-Business, E-Management and E-Learning (IC4E)*. https://doi.org/10.1145/3183586.3183607

[3] AMD. Accessed in 2020. Aparapi project. https://aparapi.github.io/

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[5] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/1375527.1375562

[6] Nathan Bell and Jared Hoberock. 2012. Thrust: Productivity-Oriented Library for CUDA. *Astrophysics Source Code Library* (2012).

[7] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction (CC)*. Springer Berlin Heidelberg.

[8] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer.* Technical Report OSU-CISRC-10/07-TR70.

[9] P. Carbone, Asterios Katsifodimos, Stephan Ewen, V. Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[10] Linchuan Chen and Gagan Agrawal. 2012. Optimizing MapReduce for GPUs with Effective Shared Memory Usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *ArXiv* abs/1802.04799 (2018).

[12] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang)*. https://doi.org/10.1145/3237009.3237016

[13] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *ACM SIGPLAN Workshop on Intermediate Representations (IR)*.

[14] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. 51, 1 (2008). https://doi.org/10.1145/1327452.1327492

[16] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. 2012. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *41st International Conference on Parallel Processing (ICPP)*.

[17] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a High-Level Language for GPUs: (Via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2254064.2254066

[18] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Asia-Pacific Programming Languages and Compilers (APPLC)*.

[19] Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. 2014. Aristotle: A Performance Impact Indicator for the OpenCL Kernels Using Local Memory. *Sci. Program.* (2014). https://doi.org/10.1155/2014/623841

[20] Juan Fumero and Christos Kotselidis. 2018. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL)*. https://doi.org/10.1145/3281287.3281292

[21] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3313808.3313819

[22] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery. https://doi.org/10.1145/3050748.3050761

[23] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ)*. https://doi.org/10.1145/2807426.2807428

[24] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*. https://doi.org/10.1145/2458523.2458526

[25] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. 2020. A Language for Describing Optimization Strategies. arXiv:2002.02268 [cs.PL]

[26] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/3168824

[27] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *International Conference on Parallel Architecture and Compilation (PACT)*.

[28] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A Code Generator for High-Performance Tensor Contractions on GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[29] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3192366.3192379

[30] Athanasios Konstantinidis, Paul H. J. Kelly, J. Ramanujam, and P. Sadayappan. 2014. Parametric GPU Code Generation for Affine Loop Programs. In *Languages and Compilers for Parallel Computing (LCPC)*.

[31] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3050748.3050764

[32] Markus Kowarschik and Christian Weiß. 2003. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies - Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*. Springer.

[33] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Wurthinger, and Hanspeter Mossenbock. 2018. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes (ManLang)*. https://doi.org/10.1145/3237009.3237013

[34] A. Munshi. 2009. The OpenCL Specification. In *IEEE Hot Chips 21 Symposium (HCS)*. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[35] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis. 2019. Towards Prototyping and Acceleration of Java Programs onto Intel FPGAs. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. https://doi.org/10.1109/FCCM.2019.00051

[36] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. 2020. Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. *The Art, Science, and Engineering of Programming* 5, 2 (2020). https://doi.org/10.22152/programming-journal.org/2021/5/8

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2499370.2462176

[38] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance Portable GPU Code Generation for Matrix Multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU)*. https://doi.org/10.1145/2884045.2884046

[39] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing (LCPC)*.

[40] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. 2015. Aparapi-UCores: A High Level Programming Framework for Unconventional Cores. In *IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC.2015.7322440

[41] Andreas Simburger, Sven Apel, Armin Größlinger, and Christian Lengauer. 2019. PolyJIT: Polyhedral Optimization Just in Time. *International Journal of Parallel Programming* (2019).

[42] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the High Road to a Low Level. *ACM Transactions on Architecture and Code Optimization (TACO)* (2015). https://doi.org/10.1145/2764907

[43] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2544137.2544157

[44] M. Steuwer, P. Kegel, and S. Gorlatch. 2011. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. https://doi.org/10.1109/IPDPS.2011.269

[45] M. Steuwer, T. Remmelg, and C. Dubach. 2016. Matrix Multiplication Beyond Auto-Tuning: Rewrite-based GPU Code Generation. In *International Conference on Compilers, Architectures, and Sythesis of Embedded Systems (CASES)*. https://doi.org/10.1145/2968455.2968521

[46] M. Steuwer, T. Remmelg, and C. Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863730

[47] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)* (2014).

[48] Xiaonan Tian, Rengan Xu, Yonghong Yan, Sunita Chandrasekaran, Deepak Eachempati, and Barbara Chapman. 2015. Compiler Transformation of Nested Loops for General Purpose GPUs. *Concurrency and Computation: Practice and Experience* (2015). https://doi.org/10.1002/cpe.3648

[49] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013). https://doi.org/10.1145/2400682.2400713

[50] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013). https://doi.org/10.1145/2400682.2400689

[51] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU Microarchitecture Through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2010.5452013

[52] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1806596.1806606

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.