# OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries

Artem Chikin, Tyler Gobran, and José Nelson Amaral

University of Alberta, Edmonton AB, Canada
{artem,gobran,jamaral}@ualberta.ca

**Abstract.** This paper presents three ideas that focus on improving the execution of high-level parallel code in GPUs. The first addresses programs that include multiple parallel blocks within a single region of GPU code. A proposed compiler transformation can split such regions into multiple, leading to the launching of multiple kernels, one for each parallel region. Advantages include the opportunity to tailor grid geometry of each kernel to the parallel region that it executes and the elimination of the overheads imposed by a code-generation scheme meant to handle multiple nested parallel regions. Second, is a code transformation that sets up a pipeline of kernel execution and asynchronous data transfer. This transformation enables the overlap of communication and computation. Intricate technical details that are required for this transformation are described. The third idea is that the selection of a grid geometry for the execution of a parallel region must balance the GPU occupancy with the potential saturation of the memory throughput in the GPU. Adding this additional parameter to the geometry selection heuristic can often yield better performance at lower occupancy levels.

## 1 Introduction

Open Multi-Processing (OpenMP) is a widely used parallel programming model that enables offloading of computation to accelerator devices such as GPUs [3]. A natural way for an experienced OpenMP CPU programmer to write OpenMP GPU code is to offload to an accelerator sections of code that contain various parallelism-specifying constructs, that are often adjacent. However, this programming style generally leads to unnecessary overheads that are not apparent to programmers unfamiliar with GPU programming and mapping of high-level OpenMP code to GPUs. Experienced GPU Programmers will instead create a common device data environment and operate on data by invoking separate kernels for each required parallel operation. This technique results in more efficient code and often reduces the overall amount of host-device data transfer, as our work will demonstrate. The main goal of this investigation is to deliver better performance for the code written by experienced OpenMP programmers that are not necessarily GPU programming experts.

When an OpenMP `target` region contains a combination of parallel and serial work to be executed in a GPU, the compiler must map these computations to the GPU's native Single Instruction, Multiple Thread (SIMT) programming model. One approach is through a technique called warp [1] specialization [2]. When specializing warps, the compiler designates one warp as the master warp and all others as a pool of worker warps. In the Clang-YKT compiler OpenMP 4 implementation, the master warp is responsible both for executing serial code and for organization and synchronization of parallel sections [7] [6]. The synchronization between parallel and serial work is implemented through named warp barriers and an emulated stack in GPU global memory for the worker warps to access the master threads state.

Figure 1 is an example of OpenMP 4 code. The pragma in line `1` establishes that the following region of code will run on the default target accelerator — assumed to be a GPU in this work, and ensures that the data specified in `map` clauses is transferred to and from the GPU, respective to the (`to`,`from`) specifiers. The pragmas in lines `2` and `6` establish the associated work as parallel within the enclosing target region. There is an implicit synchronization point at the end of each parallel region.

```
 1 #pragma omp target teams map(to: B[:S]) map(tofrom: A[:S], C[:S]) {
 2   #pragma omp distribute parallel for
 3   for () {
 4     ... // Parallel work involving A and B
 5   }
 6   #pragma omp distribute parallel for
 7   for () {
 8     ... //Parallel work involving B and C
 9   }
10 }
```

Fig. 1: Example OpenMP GPU code with multiple parallel loops in a target region.

Warp specialization introduces substantial overhead because the master and worker warps must synchronize execution when parallel region execution starts and finishes. Even when there is no sequential code between parallel regions, synchronization is required between the completion of one parallel region and the start of another.

The Clang-YKT compiler performs a transformation called *elision* that removes the warp specialization code, the master-thread stack emulation and the synchronization code, thus eliminating unnecessary overhead [7]. To be candidate for elision, a target region must contain only one parallel loop and this loop must not contain calls to the OpenMP runtime. A research question posed by

[1] CUDA terminology is used in this paper

our work is: what would be the performance effect of transforming an OpenMP 4 `target` region that contains multiple parallel regions, with or without serial code, into multiple target regions, each with a single parallel region. The goal is to enable the compiler to perform the elision transformation. Special care must be taken to avoid increasing the amount of data transfer between the host and device memory.

This paper explores two additional transformation opportunities, both applicable to any OpenMP code where parallel loops are isolated into their own target regions. The first is the overlapping of data transfer and GPU kernel execution for multiple adjacent target regions. The target regions are wrapped in a common device data environment and through memory-use analysis, a compiler can determine which data is and is not needed until or after a certain point. The second opportunity is to overlap computation with data-transfer by pipelining the loop within a single-loop parallel region in a fashion similar to iterative modulo scheduling [16]. The loop iteration space can be divided into multiple tiles, each resulting in a separate kernel launch, execution of which happens asynchronously with the data transfer for the next tile.

Finally, this target region format allows for better selection of grid geometry tailored to the contained parallel loop. Grid geometry is the number of Cooperative Thread Arrays (CTAs), also known as thread blocks, and the number of threads per CTA that the GPU uses. Grid geometry strongly affects the overall occupancy of the GPU. Tailoring this selection to a specific parallel region can have a significant effect on the performance of that region. However, a single grid geometry must be selected for an entire target region. Therefore, multiple parallel regions in the same target region cannot have individually specialized geometry for each parallel region.

In the remainder of this paper, Section 2 describes how kernel splitting enables the elision of runtime calls and barrier synchronization. Section 3 presents a sample code to demonstrate how kernel splitting is performed. Section 4 describes the implementation of asynchronous memory transfers and presents a study of their performance implications. Section 5 explains how these transfers can be used to establish a pipeline between computation and data transfers. Section 6 shows that custom grid geometry must take into consideration the potential saturation of memory bandwidth in the GPU. Section 7 presents the performance study that can be used to predict the potential benefits of the proposed transformations. Section 8 discusses other approaches to use asynchronous transfers and to adjust GPU occupancy to improve performance.

## 2 Background on Warp Specialization and Elision

GPUs' reliance on a SIMT execution model has a multitude of implications on how compilers generate GPU code from OpenMP. Where possible, parallel constructs must be mapped to a data-parallel structure in order to achieve good performance and efficiently utilize the hardware. However, full breadth of the OpenMP specification must be supported by a compliant compiler implementation. Thus, the GPU code generator must be able to handle a multitude of constructs that contain both serial and parallel code that may be nested or adjacent within a `target` region.

The compiler used in this work employs a cooperative threading model that utilizes the technique of warp specialization to generate data-parallel GPU code from parallel OpenMP regions [7]. Parallel work is performed by a collection of worker warps and coordinated by a single master warp (selected to be the last warp in a CTA). The coordination between warps is done through the use of a CTA-level synchronization primitive that allows for named barriers that apply to a compiler-specified number of warps to participate in the barrier (`bar.sync $0 $1`). When the master encounters a parallel region, it activates the required number of worker warps and suspends its own execution.

While necessary to support the full breadth of possible OpenMP constructs that can occur in target regions, as well as serial code sections and sibling parallel regions, warp specialization code-generation scheme incurs a significant amount of runtime overhead that can be avoided in select special cases. Not all kernels require the full machinery of the cooperative code-generation scheme. For target regions that are comprised solely of a single parallel loop with no nested OpenMP constructs, and no serial code, the compiler optimizes the generated code by eliding the warp specialization and runtime-managed sections of the code. This optimization results in dramatically simpler generated data-parallel code that eliminates the mentioned overheads.

Elision of the cooperative code-generation scheme and its incurred synchronization points is enabled by target region splitting. Jacob et al. describe how this elision is handled automatically by the Clang-YKT compiler, and present a performance study of elision [7]. Code that is transformed with the splitting method shown in Figure 2 creates separate target regions that are likely to satisfy all of the above conditions for elision.

## 3 Fission of Multiple-Parallel-Region Target Regions

When a target region is separated into two target regions, as shown in Figure 2, each target region is then executed as a separate kernel on the GPU and therefore data transferred for the first region is no longer present for the second region to utilize as is the case when both exist in a single-target region. Figure 2 shows how the single-target region spanning lines `1-10` in Figure 1 can be split into two separate target regions, one spanning lines `2-5` and the other lines `6-9`. The parallel region directives (lines `2` and `6` of Figure 1) are combined with the target directives (lines `2` and `6` of Figure 2), transforming each parallel region into a stand-alone target construct. To avoid extra data transfers, the newly formed target regions are enclosed in a common device data environment containing all the implicit and explicit mappings of data from the original single-target region. Only the data items specified in the data environment persist in GPU global memory across multiple target regions. The motivation for this transformation to be performed by a compiler is further reinforced by the design of the `kernels` OpenACC construct [1]. `kernels` construct definition states: "The compiler will split the code in the kernels region into a sequence of accelerator kernels", as deemed appropriate by the implementation. This design makes a strong argument for implementing the proposed transformation at the OpenMP level to further the efforts towards performance portability.

```
#pragma omp target data map(to: B[:S]) map(tofrom: A[:S], C[:S]) {
  #pragma omp target teams distribute parallel for
  for () {
    ... // Parallel work involving A and B
  }
  #pragma omp target teams distribute parallel for
  for () {
    ... //Parallel work involving B and C
  }
}
```

Fig. 2: Example OpenMP code following kernel splitting.

Furthermore, with a common device data environment, it is possible to overlap memory transfers with computation by analyzing when each data element is needed or produced. In our hand-implemented prototype for the transformation the OpenMP `target update` directive is used for these transfers, with the additional `nowait` clause added to allow for asynchronous memory transfers.

Safety measures must be taken when performing target fission, mainly to handle the presence of serial sections within the original single-target region. One concern to address is the possibility of variables being declared for the scope of the original single-target region. These variables reside in GPU memory and exist for the duration of the target region that is their scope, as a result the compiler must ensure that splitting does not interfere with any usages of them. One approach, if possible, is to move the variable declaration onto the CPU and map it to the common device data environment with an `alloc` map clause. Additional care must be taken to then mark such variables as `teams private`, to replicate the semantics of original code. Another approach is to limit the fission transformation such that all code from the declaration of the variable to its final usage resides within a single target region, though this can prevent elision.

A mitigating factor for this concern is that any such interfering declaration within the original single-target region scope must reside in a serial region at the target region scope. Variables declared inside parallel regions are assumed to be thread-local and expire when the parallel code block goes out of scope.

Another safety concern is that of serial code operating on data objects that are modified by previous parallel regions or are utilized by later parallel regions. The compiler must ensure that an updated variable is used by both the serial code and any later parallel regions on the GPU as would be the case with a single-target region wherein all code operates on the same GPU memory. One solution is to place serial code segments on the GPU in their own target regions. A drawback is paying the cost of additional kernel launch to execute serial code.

An alternative approach is to execute the serial code on the CPU, with compiler analysis ensuring that any data object used in parallel regions are transferred to and from the device as needed for correctness. These transfers can become costly if they occur frequently, but in some cases run time can be improved significantly by executing serial code on the CPU.

Therefore the kernel splitting method should be applied with caution when the original single-target region has serial code or target region scoped local variables. Such scenarios did not appear in any of the benchmarks tested and likely do not represent a large portion of OpenMP code that can benefit from splitting.

## 4 Overlapping Data Transfer and Split Kernel Execution

Overlapping data transfer with computation can be an effective strategy to increase performance. Opportunities to benefit from asynchronous data transfers may arise from the splitting of a multi-parallel-region target into multiple single-parallel region targets. To enable the pipelining of data transfers and computation, the compiler must determine the first point of use of data and also when the computation of results is completed and the data is no longer used in the target. After such analysis, a schedule can be created for the pipelining with the overlapping effectively hiding the memory transfer time.

Figures 3 and 4 illustrate how this pipelining, enabled by asynchronous memory transfers, can reduce the overall execution time. In this example, if the runtime of the two kernels are long enough, this transformation results in the costs of the asynchronous memory transfers being entirely hidden.

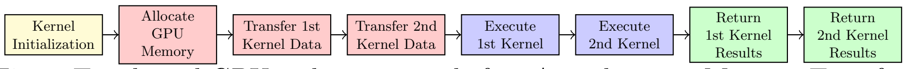

Fig. 3: Two kernel GPU code structure before Asynchronous Memory Transfer.

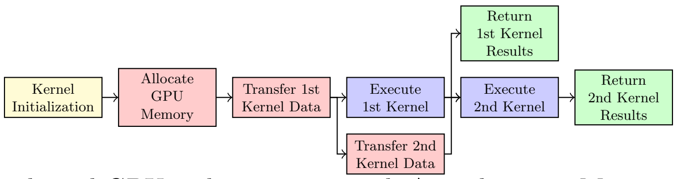

Fig. 4: Two kernel GPU code structure with Asynchronous Memory Transfer.

Execution of asynchronous memory transfers and their synchronization with kernel execution can be specified manually by a programmer, using two OpenMP 4.5 clauses: `depend` and `nowait`. An OpenMP command with a `depend` clause with an `out` attribute must finish before any command with a `depend` clause with an `in` attribute with the same value. The `nowait` clause states that the specified OpenMP task can be run asynchronously with other tasks, thus allowing the update memory transfer to occur while a target region is executing. The combination of these clauses allows for the construction of GPU code that has asynchronous memory transfers to and from the GPU while also maintaining correct computation through clearly established task dependence relations by which these asynchronous transfers must finish.

Figure 5 is an example of split target region code with asynchronous memory transfers within a common device data environment. In this example the data element `C` is not needed until the target region at line `9`, thus its mapping in the `target data` region in line `2` is only to return to the host after all work finishes. The transfer to the GPU for `C` instead begins on line `3` where it is declared asynchronous by the `nowait` clause. With the pair of `depend` clauses in lines `3` and `9` ensuring the transfer must be completed before any computation on the target region in line `9` can begin. Furthermore the array `A` can be transferred back to the host memory asynchronously as it is not used in the second target region. Thus the memory transfer of `A` back to the host is moved to line `8`, after the first target region computation and it is declared to be asynchronous.

```
 1 int a;
 2 #pragma omp targe data map(to: A[:S], B[:S]) map(from: C[:S]) {
 3   #pragma omp target update to(C[:S]) depend(out: a) nowait
 4   #pragma omp target teams distribute parallel for
 5   for () {
 6     ... // Parallel work involving A and B
 7   }
 8   #pragma omp target update from(A[:S]) nowait
 9   #pragma omp target teams distribute parallel for depend(in: a)
10   for () {
11     ... //Parallel work involving B and C
12   }
13 }
```

Fig. 5: The split OpenMP GPU code with Asynchronous Memory Transfers.

As per vendor specification, asynchronous memory transfers require that the transferred data be page-locked i.e. *pinned* on the host. A pinned page cannot be swapped out to disk and enables DMA transfers via the memory controller, bypassing the CPU. To enable asynchronous transfers, the pinning must be done through the CUDA API to allocate/free pinned memory or to pin pre-allocated heap memory. The invocation of these API functions and the actual pinning of the memory introduce additional overheads but also leads to faster memory transfers. Memory capacity constraints of the target device are not affected by the transformed kernel. The amount of data required to be present on the device at a given time is reduced in the best case, and is left unaffected in the worst.

We use the `cudaHostRegister` API to pin user-allocated memory in our experiments. The main trade-off to consider when implementing kernel asynchronous data transfers is to offset the overhead of pinning memory through faster transfers enabled by pinned memory and overlapping transfer with computation. Pinning memory also has the effect of reducing the overall memory available on the host for other processes, which can possibly stifle host computation. An important factor to consider when pinning memory is the operating system's default page size.We have found that pinning the same amount of memory was up to 10× faster on a POWER8 host with 64`KB` pages than on a x86 Haswell host with 4`KB` pages.

A synthetic experiment to illustrate the balancing of the costs and benefits of asynchronous memory transfer was designed with three simple GPU kernels ($\mathtt{k}_1$, $\mathtt{k}_2$, $\mathtt{k}_3$) that execute within a shared data environment; $\mathtt{k}_2$ modifies one data object from the CPU whose results must be returned, the object is not used by the first or third kernel. Thus, asynchronous transfer is possible both to transfer this data object to the GPU and back to the CPU. Furthermore, $\mathtt{k}_1$ and $\mathtt{k}_3$ both have enough computation to fully hide the asynchronous memory transfers. The experiment's results with a varying size of the object modified by $\mathtt{k}_2$ are shown in Figure 6. The baseline version uses unpinned memory and synchronous transfers. Four versions using pinned memory were constructed for comparison: (1) sync transfers; (2) async to/sync from; (3) sync to/async from; (4) async to/async from. The run time measured includes the time needed to allocate and free memory. The graph outlines the speedup ratio in total execution time for each of the four pinned memory versions compared to the baseline version. The horizontal axis shows both the size of the object transferred and the baseline run time measured in seconds. The results show that as the size of the transferred object increases, the additional cost of pinning memory becomes less relevant. For larger objects, even though simply pinning the memory pages yields performance gains, asynchronous memory transfers produce additional benefits.

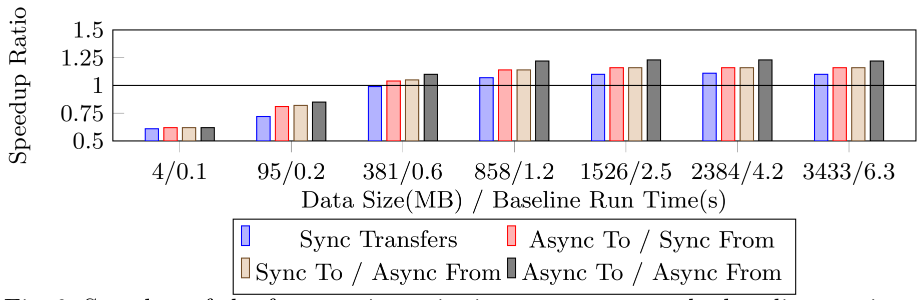

Fig. 6: Speedup of the four versions pinning memory over the baseline version.

## 5 Pipelining Data Transfer and Parallel Loop Execution

A more ambitious code transformation that utilizes the faster transfer to/from pinned memory and asynchronous communication and computation consists of breaking a singular parallel loop into multiple loops. Known as *tiling* in compiler literature, this transformation produces multiple sub-loops (tiles) which are then placed in separate target regions. After this transformation the data transfer required for the original loop may be split into several asynchronous data transfers for data elements required by the respective tiles. Ideally, each tile should use different, contiguous, large chunks of data. The goal is to overlap the transfer with computation. In the evaluation prototype OpenMP `depend` clauses are used to ensure that each data transfer is finished before the corresponding tile executes. Transmission of tile results back to the host can also be added to this pipeline. Pipelining can greatly improve the run-time performance of programs with large data transfers, when the execution time of the split loop is long enough to compensate for the overhead of setting up data transfers and pinning memory.

Figure 7 illustrates how the execution of a parallel region can be pipelined to overlap memory transfers with computation. The single parallel-loop GPU kernel is split into four tiles which allows the memory transfers required for the latter three tiles to be hidden underneath the previous tiles' execution with asynchronous transfers. Furthermore if the execution of the tiles are long enough to cover the runtime of the memory transfers then the total cost of the transfers may be as low as 1/4 of the original cost.

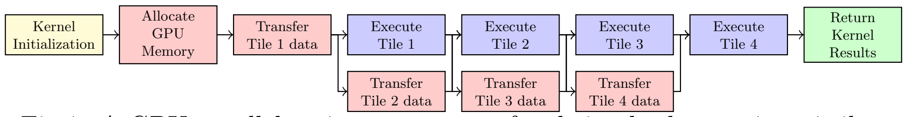

Fig. 7: A GPU parallel regions structure after being broken up into 4 tiles.

The Polybench benchmark `ATAX` is a good candidate to benefit from this transformation. The original benchmark's first parallel region, shown in Figure 8, has the majority of its runtime dependent on the memory transfer of the data object `A` to the GPU in line `1`. Figure 9 shows the code after the loop

```
1 #pragma omp target teams distribute parallel for map(to: A[:NX*NY], x
      [ :NY]) map(from: tmp[:NX]) {
2   for(int i = 0; i < NX; i++) {
3     tmp[i] = 0;
4     for(int j = 0; j < NY; j++)
5       tmp[i] = tmp[i] + A[i*NY+j] * x[j];
6   }
7 }
```

Fig. 8: First parallel region in `ATAX` before pipelining.

is divided into four tiles and the transfer of `A` split into four OpenMP `target update` calls. The first call in line `2` is not asynchronous as it must be done before the first tile execution starts. The remaining three transfers in line `6` are asynchronous and start before the preceding tile execution to overlap communication and computation. The `depend` clauses in the asynchronous transfers are needed to synchronize the end of the data transmission with the execution of the corresponding tile. Figure 9 shows a proof-of-concept manually implemented code change. A sufficiently-capable compiler should be able to apply a similar code transformation when equipped with memory access-pattern analysis to be able to separate tile data chunks, among other code safety analyses.

```
int S[4];
#pragma omp target update to(A[0:(NX/4)*NY])
for(int s = 0; s < 4; s++)
{
  if (s < 3)
    #pragma omp target update to(A[((s+1)*NX/4)*NY:((s+2)*NX/4)*NY])
        depend(out: S[s+1]) nowait
  #pragma omp target teams distribute parallel for depend(in: S[s])
  for(int i = (s*NX/4); i < ((s+1)*NX/4); i++) {
    tmp[i] = 0;
    for(int j = 0; j < NY; j++)
      tmp[i] = tmp[i] + A[i*NY+j] * x[j];
  }
}
```

Fig. 9: `ATAX` region after being broken up into four tiles for pipelining.

## 6 Custom Grid Geometry

A grid geometry defines the number of CTAs and the number of threads per CTA assigned to execute a GPU kernel. A typical GPU has a number of Streaming Multiprocessor (SM) cores that can each issue instructions for two groups of 32 threads (warps) in each cycle. An SM can maintain the state of thousands of threads in-flight, and thus can context switch execution from a warp waiting on data accesses to other warps in order to hide memory-access latency.

Each SM has a fixed-size register file, giving each CTA a register budget. At any given time the number of CTAs that can be scheduled is limited by the size of the register file. Similarly, each SM has a fixed amount of shared memory which is shared by all CTAs running on the SM. Thus, the number of CTAs simultaneously executing on an SM is also constrained by the individual CTA's shared memory use. Additional CTAs that cannot be scheduled due to these and other hardware resource limitations are queued for later execution. GPU occupancy is the percentage of available GPU threads that are used by a given kernel.

Some parallel regions with relatively low parallelism perform better when not using all available threads. A compiler can analyze parallel loops in a `target` region to select the most performant grid geometry. However, a single grid geometry has to be selected for an entire `target` region leading to a compromise that performs relatively well for all the loop nests in the region. Grid geometry specialized to each individual parallel loop, made possible by `target` region fission, can lead to significant performance improvements.

Lloyd et al. propose a compiler heuristic, based on static analysis and runtime loop tripcount data, for the selection of a grid geometry calculated by the amount of parallelism in each loop nest [13]. The heuristic takes into account the usage of registers and shared memory for each thread and CTA as it seeks to maximize the GPU occupancy. However, maximizing occupancy can often lead to far worse performance because it leads to saturation of other hardware resource, such as the memory subsystem in heavily memory-bound codes. An example of this effect occurs in the `SYRK` benchmark shown in Figure 10. At a tripcount of 4000 the best performance is achieved around 25% occupancy which is close to the Clang-YKT default of roughly 28.6% (128 CTAs on this GPU). For this case the heuristic proposed by Lloyd makes a poor choice of geometry because in seeking to maximize occupancy it does not consider memory-bandwidth saturation. Maximum occupancy produces a Unified Cache throughput of 19.742 GB/s compared to a throughput of 183.001 GB/s at the optimal occupancy of 25%; moreover, the observed Global Load Throughput of 751.8 GB/s at optimal occupancy versus 81.5 GB/s at maximal, and the respective Global Store Throughput is 91.5 and 9.9 GB/s. These metrics support the intuition that memory bus saturation can severely limit performance at high occupancy.

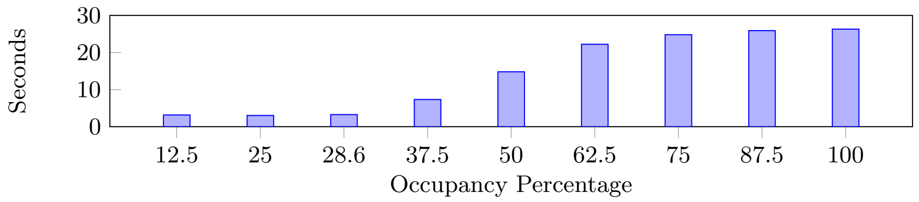

Fig. 10: Runtime results by occupancy of `SYRK` at tripcount 4000.

This exception to the grid geometry formula led to the formulation of an improved grid-geometry selection strategy for the cases where the optimal occupancy is lower than the maximum. These cases fall into the broad category of parallel regions with a high amount of parallelism exposed by the program (high parallel-loop tripcounts) and result from memory-bandwidth saturation due to a large number of memory requests. The results of this performance study allows for the classification of these cases of massively parallel memory-bound kernels into two subcategories:

**Uncoalesced Kernels** are highly memory-bound due to uncoalesced memory accesses in large tripcount parallel loops. Uncoalesced memory accesses being loads and stores to global memory where data locations accessed by adjacent threads in a warp are not grouped together closely enough, hence the warp must perform several memory accesses to satisfy all the threads in a warp. This subcategory includes the benchmarks `SYRK` with tripcount of 1000 or higher and

`COVAR` with tripcounts of 12000 or higher. `SYRK` falls into this subcategory due to the two high tripcount outer loops of its longest running parallel region being collapsed for high parallelism and an innermost loop containing an uncoalesced memory access which is performed sequentially by each thread. `COVAR` has a similar structure except without a collapse of the two outer loops and two uncoalesced memory accesses instead of one inside the inner loop. A close examination of the execution of the `SYRK` benchmark in the Nvidia Visual Profiler, reveals that the best performance is observed when the ratio between attempted memory transaction count and the memory throughput is the lowest — when the most data is transferred with the fewest requests. The grid geometry affects this ratio because more warps generate more requests when memory accesses are not coalesced.

The `SYRK` performance study shown in Figure 10 indicates that there is an opportunity to improve the grid-geometry selection by taking into consideration memory-bandwidth saturation. In a supplementary performance study we altered the ratio of requests/memory throughput in `SYRK` by adding and removing dummy uncoalesced memory accesses. This study yielded a pattern of optimal occupancy halving roughly when the number of uncoalesced memory accesses double. This insight can be used to predict the optimal occupancy for a parallel region. To analyze this pattern further a synthetic experiment was designed in which a more generalized program similar to `SYRK` was created consisting of a simple summation of the rows of $k$ different $N \times N$ matrices to produce a single matrix. The summation statement is performed within a triple-nested loop with each tripcount being 5000 and the summation involves exclusively uncoalesced memory accesses (row-major matrix accesses). The experiment was then performed with different numbers of uncoalesced memory accesses to find the optimal occupancy for each. The results of the experiment in Table 1 show the similar optimal occupancy pattern that was found in the study of `SYRK`, indicating a general pattern. This study and experiment indicates that the heuristic for grid-geometry selection introduced by Lloyd et al should be augmented to account for memory-request saturation [13].

| Number of Accesses | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Optimal Occupancy | 25% | 12.5% | 6.3% | 6.3% | 6.3% | 6.3% | 6.3% | 4.0% | 4.0% | 3.1% |

Table 1: Optimal occupancy for a massively parallel memory-bound kernel at varying numbers of uncoalesced memory accesses with tripcount 5000.

**Coalesced Kernels** have high memory utilization because of parallel loops with very large tripcounts and several memory accesses. Coalesced memory accesses are the opposite of uncoalesced and require only one access to bring over all data required by a warp of threads. This category includes the benchmarks `FDTD-2D` and `LUD` at high tripcounts. Lower occupancy results in better performance but the effect is less significant as shown in the results for the experiment study of `FDTD-2D` in Figure 11. This category should also be taken into consideration in an augmented version of the grid-geometry-selection heuristic.

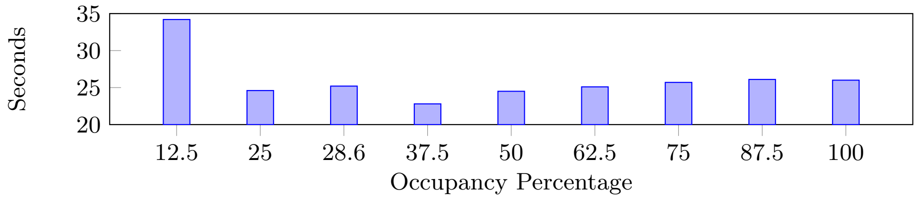

Fig. 11: Runtime results by occupancy of `FDTD-2D` at tripcount 15000.

## 7 Estimating Potential Benefits of Transformations

The goal of this experimental evaluation is to estimate the potential performance benefits of the proposed transformations to inform a design-team's decision to include them in a compiler. The results in this section are based on manually-implemented modifications to programs in the Polybench and Rodinia benchmark suites [5] [4]. Both suites have an initial OpenMP 4.0 implementation. Before performing the experiments, we modified some programs in both suites to fully utilize the GPU parallelism hierarchy with `teams` and `distribute` constructs. This experimental study uses benchmarks that contain parallel regions where the three transformations described in the paper can be applied. SPEC ACCEL benchmarks, while available to us for experimentation, contain few to none such cases. Therefore, they do not make a good case for the transformation described in this work due to their already-extensive usage of `target data` data-sharing environemnts.

All performance results reported are the average of ten runs of the program under the same conditions. Measurement variances were monitored and stayed below 1% of the average and are not reported. Two exceptions are in the execution of `SYRK` and `COVAR` that saw up to 5% variance from the average because of the effects of memory saturation. Correctness of every transformation was verified using the benchmarks' output verification mechanisms.

This experimental study uses an x86 host equipped with an Intel i7-4770 processor, 32 GiB of RAM and an NVIDIA Titan X Pascal GPU with 28 SMs and 12 GiB of on-board memory that is attached via the PCIe bus. The clock rate is locked at 80% of the nominal clock rate for the GPU to prevent variance in performance due to frequency scaling [2]. Additional experiments are performed using an IBM POWER8 (8335-GTB) host with an Nvidia P100 GPU with 60 SMs that is attached via NVLINK.

[2] Dynamic frequency scaling makes achieving consitent, reproducible results very challenging due to high variance and increased effects of device warm-up.

### 7.1 Combining Kernel Splitting with Elision Improves Performance

The effect of the transformation on performance is studied on `2MM`, `3MM`, `FDTD-2D`, `SYRK`, `COVAR`, `ATAX`, `MVT` and `BICG` applications from the Polybench benchmark suite and `SRAD` and `LUD` from the Rodinia benchmark suite. All benchmarks chosen can be logically written with a singular target region by a naive GPU OpenMP programmer. The experimental evaluation of the kernel-splitting technique includes seven different versions of each benchmark outlined in Table 2. Custom grid geometry was calculated using the heuristic by Lloyd et al. with the additional pattern for massively parallel memory-bound uncoalesced kernels described in Section 6 utilized for relevant cases [13].

| Version | Kernel Splitting | Elision | Custom Grid Geometry | Async Memory Transfer |
|---|---|---|---|---|
| Baseline | | | | |
| K | ✓ | | | |
| KE | ✓ | ✓ | | |
| KG | ✓ | | ✓ | |
| KEG | ✓ | ✓ | ✓ | |
| KA | ✓ | | | ✓ |
| KEGA | ✓ | ✓ | ✓ | ✓ |

Table 2: The experimental evaluation versions for the splitting method.

Figure 12 displays the speedup over the baseline for each benchmark and each version shown in Table 2. Asynchronous transfer is not applicable (N/A) to the `SRAD`, `FDTD-2D` and `LUD` benchmarks as they all lack memory transfers that could be performed asynchronously. In the baseline, serial code is executed between any two parallel regions and the state of the master thread is propagated to all worker threads. Kernel splitting removes the serial code and workers' update. `LUD` has few worker threads because of its low level of parallelism, thus there is little benefit to the elimination of worker updating and the cost of launching a second kernel makes `LUD` slower after splitting (version K). `LUD`'s target region is executed within a loop, which amplifies the cost of the extra kernel launch. In contrast, `FDTD-2D` and `SRAD` have far higher levels of parallelism which leads to more expensive workers' state update. Thus they benefit the most from kernel splitting.

Benefits from adding elision to splitting (version KE) vary, with `2MM`, `3MM` and `SYRK` performing poorly because the runtime's default strategy selects an inefficient grid geometry. The removal of the warp specialization and sequential code overhead makes the memory bus saturation issue more relevant leading to the lower performance. In `SYRK` the main issue is that the default occupancy is too high. In `2MM` and `3MM` the number of threads is too low to exploit all available parallelism. `FDTD-2D`, `SRAD`, and `LUD` benefit greatly from elision because they contain a large number of kernel calls, accumulating the reduction in overhead of the elided kernels over time. Moreover, the amount of parallelism and the

| Benchmark | Base Time | K | KE | KG | KEG | KA | KEGA |
|---|---|---|---|---|---|---|---|
| 2MM | 36.6s | 1.00 | 0.85 | 0.94 | 1.22 | 1.00 | 1.22 |
| 3MM | 54.8s | 1.00 | 0.85 | 0.94 | 1.22 | 1.00 | 1.22 |
| FDTD-2D | 11.8s | 1.03 | 1.23 | 0.97 | 1.37 | N/A | N/A |
| SYRK | 40.9s | 1.00 | 0.92 | 1.01 | 1.04 | 1.01 | 1.04 |
| COVAR | 54.9s | 1.00 | 1.04 | 1.02 | 1.04 | 1.00 | 1.04 |
| ATAX | 0.16s | 1.01 | 1.03 | 1.02 | 1.03 | 0.66 | 0.67 |
| MVT | 0.16s | 1.01 | 1.00 | 1.02 | 1.02 | 0.66 | 0.66 |
| BICG | 0.16s | 1.00 | 1.01 | 1.01 | 1.01 | 0.66 | 0.66 |
| SRAD | 8.90s | 1.04 | 1.45 | 1.18 | 1.48 | N/A | N/A |
| LUD | 38.1s | 0.91 | 1.57 | 0.92 | 1.57 | N/A | N/A |

Fig. 12: The speedup ratio over the baseline for each experiment evaluation of the applicable Polybench and Rodinia benchmarks run at a tripcount set to 9600. `SRAD` executes on a 512by512 image with the encompassing iteration loop performed 9600 times. `LUD` operates on a 9600by9600 matrix.

compute-bound nature of the kernels in these benchmarks suit the compiler's default grid geometry selection strategy.

Asynchronous memory transfer (version KA) by itself produces either negligible benefits or performance degradation. The degradation for `ATAX`, `MVT` and `BICG` results from the small size of data objects making the cost of pinning the data for transfer far greater then any hidden transfer cost and the short length of the benchmarks emphasizes this.

In general, significant performance improvements are achieved by the kernel-splitting technique combined with elision for the given benchmarks. Furthermore, any poor performance can be mitigated by additional procedures such as tuning grid geometry that are only available once the splitting technique is applied.

### 7.2 Elision Amplifies Benefits of Custom Grid Geometry

`SYRK` and `COVAR` are the benchmarks most affected by the grid-geometry selection. Both are highly memory-bound because they contain frequently executed uncoalesced memory accesses (`SYRK` has one, and `COVAR` has two) and as a result they both have lower than maximum optimal occupancies that produce large performance improvements. `COVAR` only has a lower than maximum optimal occupancy at higher tripcounts as it lacks the high parallelism of `SYRK`. These benchmarks' optimal occupancies decrease as the number of memory accesses rise with higher tripcounts, the optimal percentages following the optimal-occupancy trend outlined in Table 1.

Further experimental evaluation of `SYRK` and `COVAR` at multiple tripcounts for both a base unsplit version and a KEG version illustrates the effects of varying the grid geometry. The optimal occupancy, determined by the grid geometry, changes with the amount of parallelism for both benchmarks. For `SYRK` the optimal occupancy is 25% at lower tripcounts and 18.75% at higher tripcounts, while for `COVAR` the optimal is the default heuristic presented by Lloyd et al.

that has the occupancy slowly grow towards the maximum for lower tripcounts when parallelism is low, with higher tripcounts having an optimal occupancy of 12.5% [13]. To illustrate this shift of optimal occupancy the experimental results shown in Figure 13 present the speedups of the two benchmarks' KEG version over the baseline for both of their optimal occupancies. The large improvement for `SYRK` at tripcount 3000 matches a similar effect in other programs with collapsed parallel loops that is caused by a sufficiently high parallelism. This performance is due to a combination of a GPU code that was simplified by elision and low impact of memory bus saturation because of still relatively low parallelism.

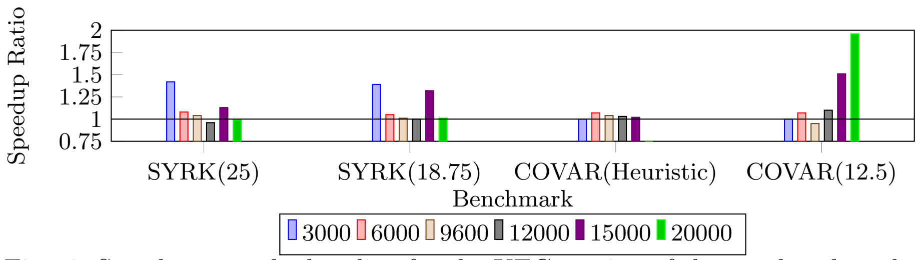

Fig. 13: Speedup over the baseline for the KEG version of the two benchmarks at their two improved occupancies with varying tripcounts. Heuristic refers to that by Lloyd et al. [13]

`2MM`, `3MM` and `FDTD-2D` present slight performance degradation when only custom grid geometry is applied (version KG) as with all three benchmarks the custom grid geometry is set to achieve full occupancy of the GPU SM's. With elision this is optimal but without elision additional warps are added on top of the full occupancy for the master warps in each CTA. As a result the executed kernels request more warps then can be active on the GPU concurrently, thus additional scheduling of the warps is performed by the GPU to ensure all warps execute. This scheduling causes overhead that result in worse performance for the three benchmarks compared to when only kernel-splitting is applied. In comparison significant performance improvements for `SRAD` with version KG come from the high amount of computation compared to memory accesses in the program which take advantage of increased GPU occupancy from the heuristic.

The improvements brought by custom grid geometry (version KG) are amplified when combined with elision (version KEG) because the simplified execution for elided code better utilizes an optimized number of CTAs in terms of memory utilization and computation ability. Thus, the version KEG produces the best performance through this amplification of the benefits of elision and custom grid geometry.

Finally asynchronous transfers do not interact with grid geometry in any meaningful way, as such the KEGA results are only presented for completeness.

### 7.3 Pipelining Improves Performance for High Trip Counts

The pipelining transformation requires that a parallel region be broken into sub-loops that process separate data chunks of sufficiently large size to justify the pipeline. Thus only the Polybench benchmarks `ATAX`, `GESUMMV` and `GEMM` are suitable for pipelining. Memory transfers to/from the GPU are dominant for the execution time for `ATAX` and `GESUMMV` resulting in significant improvements from pipelining. These improvements increase with the number of iterations as the additional computation amortizes the cost of pinning memory pages. Pipelining transfers plays a minor role in `GEMM` because kernel execution is dominant, instead the restructuring of computation caused by splitting the kernel in four improves performance. Each of the four resulting kernels have a quarter of the parallelism of the original kernel and thus a higher number of loop iterations can be executed before memory bandwidth saturation requires reduction in occupancy. A similar effect occurs for `ATAX` and `GESUMMV` but due to the dominant memory transfers the effect on performance is minimal.

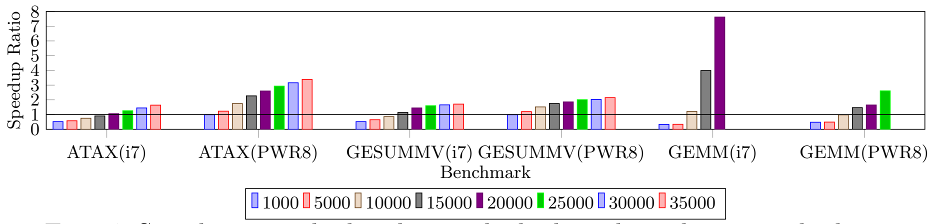

Fig. 14: Speedup over the baseline with the kernel pipelining method on applicable Polybench benchmarks at varying tripcounts. GEMM is missing sizes due to time constraints.

For the experimental evaluation of kernel pipelining on the benchmarks `ATAX`, `GESUMMV` and `GEMM`, the baseline is a KE version of the benchmarks with non-pinned memory for all data objects. This baseline is compared to a version that has data transfer pipelined into four tiles with all pipelined data objects pinned. Both versions utilize the default Clang-YKT grid geometry formula. The evaluation is run on two machines: the Intel i7-4770 described above and a POWER8 host with a P100 GPU. The results in Figure 14 show significant improvements in performance with kernel pipelining for sufficiently large tripcounts. The POWER8 speedup is far larger because it uses 64KB pages compared to the 4KB pages in the Intel i7. The larger page size greatly reduces the cost of pinning memory. `GEMM` sees significant performance improvement because each individual tile processes a smaller chunk of data, thus allowing for higher utilization of the device without hitting the memory subsystem saturation performance barrier. In comparison, the benefits for `ATAX` and `GESUMMV` emerge from hiding transfer cost. At lower tripcounts the transformation degrades performance because there is not enough parallelism in the tiles to utilize as many GPU SMs, and the overhead of pinning memory and initializing additional kernels is not overcome.

In a second version of the experiment, on the x86 machine, the optimal occupancy at every tripcount for each kernel is applied to remove the influence of memory saturation. For `GESUMMV` and `GEMM` the optimal is 12.5% occupancy for the baseline and the Clang-YKT default formula for the pipelined version. While `ATAX` has an optimal occupancy of 18.75% for the baseline and the default formula for the pipelined version. Figure 15 shows this experiment's results with significantly better performance for the baseline that lowers the speedup from the pipelined version. However the pipelining still shows benefits due to pipelined memory transfers and later memory saturation at higher tripcounts of the benchmarks.

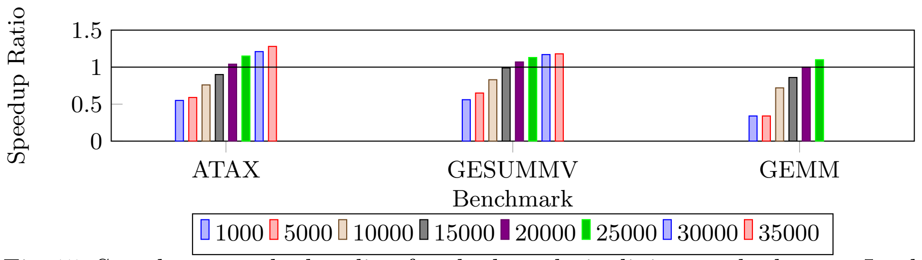

Fig. 15: Speedup over the baseline for the kernel pipelining method on an Intel i7 machine at varying tripcounts with optimal occupancy applied to all kernels.

## 8 Related Work

**Asynchronous transfers** are used for BigKernel, by Mokhtari et al., which breaks up a kernel into smaller kernels and pipelines memory transfer in a similar fashion to our kernel pipelining process [14]. BigKernel is a coding framework wherein a memory transfer that would be too large for the available GPU memory is partitioned into segments which are then transferred onto the GPU as needed. The data segments are laid out by BigKernel in host memory by analyzing the GPU kernel and organizing all data items that are used into a prefetch buffer in the order of their access by the GPU threads, creating more coalesced memory accesses in the GPU as memory accessed at the same time is placed beside each other. As with our pipelining method these transfers are performed asynchronously and are overlapped with unrelated kernel computation. However, BigKernel requires a programmer to specify different GPU calls as opposed to the compiler transformation that we propose. Furthermore, BigKernel focuses on very large data and thus its design requires double the original number of threads for a kernel, with half the threads utilized for calculating prefetch addresses. These additional overheads are not present in our method.

A common approach to pipeline GPU execution uses double buffering. Komoda et al. present a OpenCL library that optimize CPU-GPU communication by overlapping computation and memory transfers based on simple program-descriptions written by the programmer [10]. Komoda's work is limited to pipelining memory transfers with existing GPU kernels, and requires programmer specification. Our approach, in contrast, creates multiple kernels out of a single description of a GPU program (a single `target` region) to enable pipelining.

**GPU occupancy** is the focus of Kayıran et al.'s DYNCTA, a dynamic solution similar to ours that accounts for memory saturation by reducing occupancy [8]. DYNCTA analyzes each GPU SM's utilization and memory latency during execution and adjusts the occupancy within the SM to avoid memory-bandwidth saturation by keeping occupancy lower than the maximum. Changing the defined grid geometry for a kernel is impossible, as a result occupancy adjustment is achieved by assigning additional CTAs to a SM that have already been allocated to the kernel at the start of execution. Once assigned to an SM a CTA cannot be removed, as a result adjustment is performed by prioritizing or deprioritizing CTAs. A prioritized CTA has any available warps executed before a deprioritized CTA's warps, as a result with memory intensive programs wherein all warps stall on memory accesses the deprioritized CTAs will eventually be utilized after all prioritized warps stall. Performance is improved by having the occupancy just below the threshold where memory saturation causes negative effects, ensuring the SM remains utilized while avoiding the punishing effects of memory saturation. Analysis is recorded in two hardware counters within each SM, that record how long each SM has been under utilized and how the often the SM has stalled due to memory access waiting. Sethia et al. describe a similar approach with Equalizer, a heuristic that dynamically adjusts the number of CTAs based on four hardware counters [18]. Lee et al. propose a slightly different strategy with "Lazy CTA Scheduling" (LCS) wherein the workload of an initial prioritized CTA is calculated by a hardware performance counter and that data is used to calculate an improved number of CTAs for each SM [11]. In contrast, our grid geometry proposal is based on a simple hybrid analysis with a low runtime cost and is suitable for simple GPU kernels, which represent the majority of benchmarks we have tested. The benefits of a simple heuristic approach over heavyweight dynamic mechanisms, as outlined by Lloyd et al. allow for a practical deployment in a production system, even if it does sacrifice some optimility [13].

Sethia et al. present the Mascar system, which approaches memory saturation by prioritizing the accesses of a single warp instead of a round robin approach [17]. The single warp starts computation earlier to help hide the latency of other accesses, with the scheduler additionally prioritizing warps with computation over memory-accessing warps when memory saturation is detected. A queue for failed L1 cache access attempts is also added to the GPU hardware, holding the accesses for later execution, it prevents warps from saturating the cache controller with repeated access requests so that other warps can attempt their accesses. Mascar requires hardware design and warp scheduling changes. In contrast, our custom grid geometry based on static analysis is far less intrusive.

Other dynamic approaches include Oh et al.'s APRES, a predictive warp scheduler that prioritizes the scheduling of groups of warps with likely cache hits [15]. Kim et al. suggest an additional P-mode for warps waiting on long memory accesses wherein later instructions that are independent of the long accesses are pre-executed while any dependent operations are skipped [9]. Lee et al.'s CAWA reduces the disparity in execution time between warps by providing the slower running warps with more time to execute and a reserved area of the L1 cache [12]. All these approaches have additional run time costs when compared to a static analysis and compile-time selection of custom grid geometry and redistribution of work across multiple kernels enabled by pipelining. Furthermore, warp scheduling approaches and custom grid geometries are complimentary and can be combined.

## 9 Conclusion

This paper puts forward the idea of splitting a singular OpenMP target region of GPU code with multiple parallel regions into multiple target regions each with a singular parallel region. The experimental evaluation using Polybench and Rodinia benchmarks indicates that there can be non-trivial performance gains from implementing this idea in future compilers targeting OpenMP `4.x`. Additionally the evaluation indicates that combining kernel splitting with synchronization elision and support for asynchronous memory transfers (with OpenMP 4.5) would lead to even more significant performance.

The study of grid geometry indicates that there is scope to improve existing grid-geometry selection strategy by considering the saturation of the GPU memory bandwidth due to uncoalesced memory accesses or to data-intensive parallel loop nests. This problem was recognized before with both dynamic runtime solutions and hardware changes proposed. However, the solution proposed here based on static analysis and compiler action is simpler, effective, and has lower overhead.

This paper also studies the performance effect of pipelining memory transfers with kernel execution when there is sufficient data. Both kernel splitting and loop tiling can be used to enable pipelining. The results indicate that the performance gains can be significant especially in machines with larger page sizes such as the POWER architecture.

## Acknowledgements

This research was supported by the IBM Canada Software Lab Centre for Advanced Studies (CAS) and by the National Science and Engineering Research Council (NSERC) of Canada through their Collaborative Research and Development (CRD) program and through the Undergraduate Student Research Awards (USRA) program.

## References

1. The openacc application programming interface. `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf`.
2. Bauer, M., Cook, H., and Khailany, B. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *High Performance Computing, Networking, Storage and Analysis SC* (Seattle, WA, USA, 2011), pp. 1–11.
3. Board, O. A. R. Openmp application programming interface. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)* (2009), pp. 44–54.
5. Chikin, A. Unibench for openmp 4.0. `https://github.com/artemcm/Unibench`.
6. Jacob, A. C., Eichenberger, A. E., Sung, H., Antao, S. F., Bercea, G. T., Bertolli, C., Bataev, A., Jin, T., Chen, T., Sura, Z., Rokos, G., and O'Brien, K. clang-ykt source-code repository. `https://github.com/clang-ykt`.
7. Jacob, A. C., Eichenberger, A. E., Sung, H., Antao, S. F., Bercea, G. T., Bertolli, C., Bataev, A., Jin, T., Chen, T., Sura, Z., Rokos, G., and O'Brien, K. Efficient fork-join on GPUs through warp specialization. In *High Performance Computing HiPC* (Jaipur, India, 2017), pp. 358–367.
8. Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Parallel Architectures and Compilation Techniques PACT* (Piscataway, NJ, USA, 2013), pp. 157–166.
9. Kim, K., Lee, S., Yoon, M. K., Koo, G., Ro, W. W., and Annavaram, M. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *High Performance Computer Architecture HPCA* (Barcelona, Spain, 2016), pp. 163–175.
10. Komoda, T., Miwa, S., and Nakamura, H. Communication library to overlap computation and communication for opencl application. In *Parallel and Distributed Processing Symposium Workshops IPDPSW* (Shanghai, China, 2012), pp. 567–573.
11. Lee, M., Song, S., Moon, J., Kim, J., Seo, W., Cho, Y., and Ryu, S. Improving GPGPU resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture HPCA* (Orlando, FL, USA, 2014), pp. 260–271.
12. Lee, S.-Y., Arunkumar, A., and Wu, C.-J. CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *International Symposium on Computer Architecture (ISCA)* (Portland, Oregon, 2015), ACM, pp. 515–527.
13. Lloyd, T., Chikin, A., Amaral, J. N., and E.Tiotto. Automated GPU grid geometry selection for OpenMP kernels. In *Workshop on Applications for Multi-Core Architectures* (September 2018), WAMCA 2018. Pre-print Manuscript. Available: `https://webdocs.cs.ualberta.ca/~amaral/papers/LloydWAMCA18.pdf`.
14. Mokhtari, R., and Stumm, M. Bigkernel – high performance CPU-GPU communication pipelining for big data-style applications. In *International Parallel and Distributed Processing Symposium IPDPS* (Phoenix, AZ, USA, 2014), pp. 819–828.
15. Oh, Y., Kim, K., Yoon, M. K., Park, J. H., Park, Y., Ro, W. W., and Annavaram, M. APRES: improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH Computer Architecture News 44*, 3 (2016), 191–203.

16. Rau, B. R. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (New York, NY, USA, 1994), MICRO 27, ACM, pp. 63–74.
17. Sethia, A., Jamshidi, D. A., and Mahlke, S. Mascar: Speeding up GPU warps by reducing memory pitstops. In *High Performance Computer Architecture HPCA* (San Francisco, CA, USA, 2015), pp. 174–185.
18. Sethia, A., and Mahlke, S. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *International Symposium on Microarchitecture MICRO* (Cambridge, UK, 2014), pp. 647–658.