# Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications

Arian Maghazeh*, Sudipta Chattopadhyay†, Petru Eles* and Zebo Peng*
*Department of Computer and Information Science, Linköping University, Sweden
†Singapore University of Technology and Design, Singapore

***Abstract*—We present a software approach to address the data latency issue for certain GPU applications. Each application is modeled as a kernel graph, where the nodes represent individual GPU kernels and the edges capture data dependencies. Our technique exploits the GPU L2 cache to accelerate parameter passing between the kernels. The key idea is that, instead of having each kernel process the entire input in one invocation, we subdivide the input into fragments (which fit in the cache) and, ideally, process each fragment in one continuous sequence of kernel invocations. Our proposed technique is oblivious to kernel functionalities and requires minimal source code modification. We demonstrate our technique on a full-fledged image processing application and improve the performance on average by 30% over various settings.**

## I. Introduction

A GPU application often involves several kernels that are inter-dependent in a fairly complex fashion. In such an application, a holistic view of the application graph can expose opportunities for performance enhancement often overlooked by conventional kernel-specific optimizations. Our tool, named KTILER, precisely takes this approach. The core idea is to leverage the GPU L2 cache (shared by all multiprocessors in the GPU) to accelerate the kernels whose performances are limited by memory latency. Technically, KTILER splits a kernel with large inputs into multiple smaller sub-kernels. The exact nature of splitting (i.e., which kernels to split and the number of sub-kernels and their input sizes) is determined so that it increases the odds of finding the intermediate data in the cache. The generated sub-kernels are then scheduled such that data dependencies are respected. KTILER is mostly automatic and works for various GPU-based applications.

*GPU execution model.* GPUs are massively parallel processors capable of running thousands of threads in parallel. Using the CUDA terminology, a *thread* is the smallest execution unit and a *kernel* is a program that runs on the GPU. A group of 32 threads form a *warp*. Threads in a warp execute instructions in the lock-step fashion. A group of warps are organized into a one-, two-, or three-dimensional *thread block*. A block is executed on a GPU multiprocessor. A limited number of concurrent blocks can reside on one multiprocessor. Blocks execute independently from each other and are further organized into a one-, two-, or three-dimensional *grid* of thread blocks. The minimum grid size is one block.

*Motivational example.* Consider the example in Figure 1(a), where an image of $256 \times 256$ pixels is first converted to a grayscale image, by GPU kernel `A`, and then scaled down to an image of $128 \times 128$ pixels, by kernel `B`. Both kernels have two-dimensional grid and block sizes. For example, kernel `A` is launched as a grid of $8 \times 32$ blocks, where each block contains 32×8 threads, specified as `A` $<<<(8\times32),(32\times8)>>>$ in CUDA. Threads in one block process a small fragment of the input image (e.g., the yellow and red tiles on `in` and `intm` images corresponding to blocks (0,0) of kernels `A` and `B`, respectively) and produce a small fragment of the output image (e.g., yellow and red tiles on `intm` and `out` images, respectively). In the normal execution mode, kernel `A` is launched first and runs to completion before it hands out `intm` as the input to kernel `B`. In this mode, the probability of finding `intm` pixels in the cache (as input to kernel `B`) diminishes rapidly as the size of image `in` exceeds the cache size.

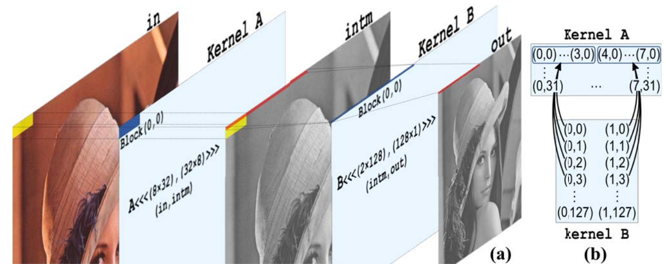


Fig. 1. (a) Mapping two GPU kernel blocks, from two consecutive kernels, to their input and output pixels. (b) Block dependencies between the two kernels

Alternatively, consider a scenario where two sub-kernels are scheduled such that blocks (0,0)...(3,0) of kernel `A` are processed before blocks (0,0)...(0,3) of kernel `B`. In such a scenario, it is likely that the threads of sub-kernel `B` will discover the fragment of `intm` image, as their input, in the cache. This style of scheduling can be repeated until the entire input image is processed. If we systematically split the blocks of kernels `A` and `B` among sub-kernels, considering the cache size, and subsequently interleave the sub-kernels, then most of them are likely to find their respective inputs in the cache.

*How* KTILER *differs from the state-of-the-art?* Towards diminishing the memory bottleneck, existing hardware-based solutions [1], [2], [3] introduce extra hardware-design complexity, whereas existing software-based solutions [4], [5], [6], [7] are either restricted to specific functionality or focus on optimizing GPU kernels in isolation. KTILER does not require additional hardware and can be used for most commodity GPU-based systems. Moreover, our proposed approach neither requires prior knowledge about an application nor is it restricted to kernel-level optimization. Concretely, KTILER provides a flexible mechanism to optimize GPU applications with multiple inter-dependent kernels.

Tiling has been used to enhance the performance of GPU

978-3-9819263-2-3/DATE19/©2019 EDAA 

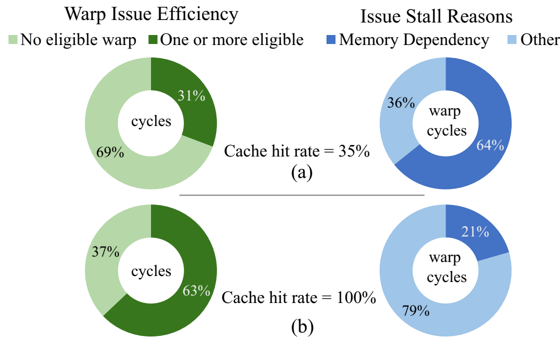


Fig. 2. Kernel performance at (a) default, and (b) $\frac{1}{32}$ of the default grid size

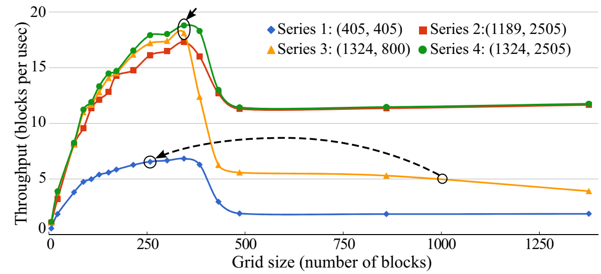


Fig. 3. Performance of the Jacobi kernel under different (GPU, MEM) frequency (MHz) configurations

kernels. In [8], the authors introduced a technique to split concurrently executing kernels at runtime to improve resource utilization. However, this technique requires the kernels to be data independent. In [6], the authors proposed a tiling-based technique to effectively use the scratchpad memory for stencils. However, this technique is only applicable to stencils, unlike KTILER, which is unaware of the kernel functionality.

*Contributions.* We introduce KTILER, a system-level tool to improve the performance of cache-sensitive GPU-based applications. We

1) propose a function-oblivious optimization technique that leverages the shared GPU L2 cache to improve the performance of cache-sensitive kernels (Section IV-C);
2) develop a tool to construct the dependency graph and obtain memory footprints of arbitrary GPU-based applications at block level (Section IV-B); and
3) apply KTILER to a full-fledged image processing application composed of over a thousand kernels (Section V).

## II. TILING IMPACT ON PERFORMANCE

Tiling improves the cache hit rate by eliminating cold misses. These are misses that occur as data is accessed for the first time by a thread. The impact of the grid size on cache performance and the difference between the minimum and maximum cache hit rates are kernel specific attributes. For example, in a kernel with a high data locality per thread (e.g., a convolution filter), one cold miss is followed by multiple hits; therefore, the minimum and maximum hit rates are both high and the gap is small. Conversely, in a kernel with a low data locality per thread (e.g., a scan algorithm), cold misses have a large weight in determining the hit rate and the gap is big. The first condition that a kernel must satisfy to be a good fit for tiling is to have a reasonably large gap between the cache hit rates at the default and at the minimum grid sizes; in other words, there must be room for improvement by tiling.

Next, performance must be limited by memory accesses; tiling is about reducing access time, and it is not useful if the bottleneck lies on other factors (e.g., thread synchronization).

The third condition is that the data dependencies of the kernel blocks must not depend on the input value. This is required to be able to offline determine the dependencies of the consumer (kernel) blocks on the producer blocks.

As a case study, we compare the performance of the Jacobi kernel, used as part of an optical flow application (Section V), at the default application grid size with a sub-kernel of $\frac{1}{32}$ of the default size. Figure 2 shows the performance metrics of the two kernels (provided by the NVIDIA profiler tool). The kernel fulfills the three tiling conditions: the cache hit rate goes up from 35% for the original kernel to 100% for the sub-kernel; memory accesses make the major performance bottleneck with memory-dependency stalls contributing to 64% of the total stalls; and the memory access pattern is not input dependent. Observe that the increased cache hit rate doubles the warp issue efficiency (a metric that indicates the availability of eligible warps per cycle across the GPU) and significantly reduces the memory dependency stalls.

Finally, the achieved performance gain must outweigh the tiling overhead. There are two potential sources of overhead: *(i)* tiling may harm the GPU utilization by reducing occupancy or increasing work imbalance across the GPU multiprocessors; *(ii)* it increases the overall inter-launch gap, which is the idle time between consecutive kernel launches, by requiring multiple sub-kernel launches. The latter, however, is not an intrinsic characteristic of the kernel and can be mitigated; for example, by improving the device driver.

In short, a kernel is suitable for tiling if it satisfies the three tiling conditions and its performance gains outweigh the tiling overhead. Figure 3 shows the relation between the grid size of the Jacobi kernel and its throughput under four different GPU/memory frequency configurations. In all series, throughput initially increases with the grid size, as a result of a higher GPU utilization, and then decreases up to a certain point as cache performance degrades. There are several points worth mentioning: First, instead of processing a thousand blocks in one kernel launch under series-3 configuration, we can split the workload into four sub-kernels of 250 blocks under series-1 configuration. As a result, not only does the throughput increase from 5 to 6.5 blocks per usec, but also the system power decreases due to significantly lower GPU/memory frequencies at series-1. Next, note that the maximum throughputs of series-3 and series-4 (obtained at grid size 344) are almost the same, even though the memory frequency in series-3 is less than one-third of that in series-4. This is because at this grid size the global memory requests are responded by the L2 cache and are not directed to the global memory. However, as the grid size increases, the hit rate decreases, until the throughput of series-3 drops to half of that of series-4.

According to our study, some other kernels that respond well to tiling include reduction, scan (Hillis Steele), bitonic sort on large arrays, matrix multiplication on arrays with special dimensions, matrix transpose, and Black–Scholes.

## III. Problem Formulation

The problem of tiling reflects the following question: "Given an application graph (i.e., a graph whose nodes capture GPU kernels and edges capture data dependencies) and an input size (which determines the number of blocks in each kernel), how to split the kernels and schedule the sub-kernels such that the overall execution time of the application is minimized?" Our approach is based on using the GPU L2 cache.

Let $N \subset \mathbb{N}$ be the set of application nodes, where each node represents a kernel. Each kernel $v \in N$ is split into a set of sub-kernels $S_v$. We denote the $i$-th sub-kernel of kernel $v$ as $\delta_v^i$. A *schedule* is defined as a partial order $\prec_{sch}$ on the set of all sub-kernels in the application graph, say $\mathcal{G} = \bigcup_{v \in N, \delta_v^i \in S_v} \delta_v^i$. For any two sub-kernels $\delta_x^y, \delta_{x'}^{y'} \in \mathcal{G}$, we have $\delta_x^y \prec_{sch} \delta_{x'}^{y'}$ iff $\delta_x^y$ is scheduled *before* $\delta_{x'}^{y'}$. The sub-kernels induce a partition on the set of blocks processed by a kernel: Let us assume $B_v$ is the set of blocks processed by kernel $v$ and $B_{\delta_v^i}$ is the set of blocks processed by its $i$-th sub-kernel $\delta_v^i$. Then, for any kernel $v$ and $\delta_v^i, \delta_v^j \in S_v$, we have $B_{\delta_v^i} \cap B_{\delta_v^j} = \phi$ and $\bigcup_{\delta_v^i \in S_v} B_{\delta_v^i} = B_v$. We note that a valid schedule must adhere to the data dependencies among all sub-kernels. We capture the dependency between an arbitrary pair of sub-kernels $\delta_x^y$ and $\delta_{x'}^{y'}$ via the notation $\delta_x^y \xrightarrow{k} \delta_{x'}^{y'}$. If $\delta_x^y \stackrel{k}{\nrightarrow} \delta_{x'}^{y'}$, then sub-kernels $\delta_x^y$ and $\delta_{x'}^{y'}$ are *independent*. $\delta_x^y$ depends on $\delta_{x'}^{y'}$ iff any block in $B_{\delta_x^y}$ depends on any block in $B_{\delta_{x'}^{y'}}$. Dependencies only exist between blocks of different kernels; within a kernel, blocks are independent: $\delta_x^y \stackrel{k}{\nrightarrow} \delta_{x'}^{y'}$ if $x = x'$. For any pair of blocks $B$ and $B'$, $B$ depends on $B'$ iff a thread in $B$ reads a memory address previously written by a thread in $B'$.

GPUs potentially allow for executing blocks from several kernels concurrently, provided that there are enough resources on the streaming multiprocessor. Otherwise, the GPU executes the kernels successively according to the order of invocations. In practice, however, even a kernel with a low grid size often fully occupies (but not necessarily fully utilizes) the multiprocessor. Therefore, in this work we assume that sub-kernels are run successively and the final schedule is a total order on the set of all sub-kernels. The total execution time ($ET$) of the application graph is then obtained by summing up the execution time of all sub-kernels: $ET(Sch) = \sum_{v \in N, \delta_v^i \in S_v} ET(\delta_v^i)$. Our objective is to find the schedule with the minimum execution time, $Sch_{opt} = argmin_{Sch} ET(Sch)$.

## IV. Methodology

We first provide an overview of our technique and then elaborate on each module in the subsequent sections.

### A. High-Level System View

KTILER consists of two main modules: a *block analyzer*, and a *scheduler*. The block analyzer provides block related information to the scheduler, including *(i)* the dependency graph, which captures data dependencies among all kernel blocks, and *(ii)* the list of all memory lines that are accessed by each block. The set of memory lines is used to obtain the memory footprint of sub-kernels. KTILER takes the application source code and a sample set of application inputs with a given size. We note that different input sizes may lead to different schedules because the number of blocks often depends on the input size. However, inputs of the same size result in similar grid sizes and identical block dependencies. Thus, for a given input size, it is sufficient to generate the schedule only once.

KTILER integrates user-provided information. The information consists of platform-specific performance characteristics of the kernels, such as the execution times of the kernel with different tilings and without it, and the degree of cache sensitivity of the kernel with respect to its inputs (Section IV-C).

Given the inputs, KTILER uses a heuristic to generate a schedule involving all sub-kernels. The schedule is then enforced at runtime by slightly modifying the source code [9].

### B. Block Analyzer

This module provides the block dependencies, used to retain functional correctness, and the list of the memory lines accessed by each block, used to attain a high cache performance.

*1) Constructing Dependency Graph:* We construct a block-dependency graph that captures the data dependencies between blocks. Using such a graph, we ensure that a sub-kernel is executed only after all its block dependencies are met. Note that the application graph (like the one in Figure 4) just captures the coarse-grained dependencies at the kernel level and pessimistically assumes that all sub-kernels of a consumer kernel can be executed only after the execution of all sub-kernels in the producer. Therefore, the default application graph is insufficient for tiling. For example, in Figure 1(b), blocks (0,0)…(0,3) of kernel `B` only depend on blocks (0,0)…(3,0) of kernel `A` and may be processed immediately after them—and before other blocks of kernel `A`.

We construct the block-dependency graph in two stages: First, we record a trace of memory accesses by all threads during an execution of the application. For each memory access, we record the effective memory address, the type of access (e.g., load, store and atomic), target memory type (e.g., global, shared or texture memory), and access width (e.g., a 4-byte operation). This step involves running the application on the GPU and using the host (CPU) to record the trace. In the second step, we process the recorded trace and construct the dependency graph based on the block dependency relation. We note that the memory trace is obtained for each thread. Hence, the recorded trace contains fine-grained information about GPU memory addresses accessed by each block. The second step is performed entirely on the host.

To obtain the memory trace of an application, we use the SASSI instrumentation tool [10]. SASSI allows developers to inject user-level instrumentation code at specific instructions or instruction types. A typical instrumentation involves the following tasks: *(i)* Compiling the application with SASSI-augmented compiler. This, in turn, enables callbacks to instrumentation handlers before or after the instrumented instruction. *(ii)* Defining the instrumentation handler, compiling it, and linking it with the application [10].

*2) Providing Block Memory Lines:* The block analyzer provides a list of the memory lines accessed by each block during execution. This list is used by the scheduler to calculate the memory footprint for each sub-kernel (Section IV-C2). A host routine uses the memory trace to generate the list.

### C. Block Scheduler

There are two pieces of user-provided information, which KTILER uses to account for the performance characteristics of each kernel on the specific platform. (The exact use of the information will become clear in Section IV-C2.)

*Edge weights.* Each edge in the application graph is associated with a weight that reflects the cache-sensitivity of the consumer node with respect to the input corresponding to that edge. The weight is equal to the maximum amount of time that can be saved if the corresponding input data reside in the cache. If a node is non-tileable (Section II), we set the weights of its input edges to zero [9].

*Performance tables.* These provide estimations regarding the execution time of a kernel. Kernel execution time is defined by two factors: the number of blocks (i.e., grid size) and the inputs that are provided via tiling (and will likely be present in the cache). To account for the second factor, each node has a table for every combination in which a set of inputs are provided by tiling. To reduce the number of tables, one may consider only the inputs whose corresponding edge weights are larger than a predefined threshold [9]. Each table contains the execution times for several grid sizes—as provided by the user. For the missing points, the duration is obtained by interpolation. KTILER obtains the execution time of a kernel by referring to the appropriate table based on the current tiling and looking up the table by using the grid size as the index.

*1) A Two-Phase Approach:* The choice of which kernels to tile and how to tile them involve conflicting design constraints. For example, tiling a kernel into large sub-kernels may increase the GPU utilization, but it also increases the memory footprints of the sub-kernels, which may lead to a lower cache performance.

We break down the scheduling problem into two phases: *(i)* node partitioning (coarse-grained scheduling), where nodes in the application graph are grouped into clusters, and *(ii)* cluster tiling (fine-grained scheduling), which involves splitting the nodes in each cluster into sub-kernels and scheduling the sub-kernels. In the following, we describe these two phases.

*Node partitioning.* Let $N \subset \mathbb{N}$ be the set of nodes in the application graph $\mathcal{T}$. A *cluster* $C$ is defined as a connected subgraph over $\mathcal{T}$. For the sake of brevity, we capture a cluster $C$ via the set of nodes defining the respective sub-graph. Therefore, $C \subseteq N$. A set of clusters $\bigcup_{i=1}^{m} C_i$ defines a partition over $\mathcal{T}$ iff *(i)* $\bigcup_{i=1}^{m} C_i = N$, and *(ii)* for any $i, j \in [1, m]$, either $C_i \cap C_j = \emptyset$ or $C_i = C_j$. We define a total order $C_i \prec_C C_j$ between a pair of arbitrary clusters, which means that all the nodes in cluster $C_i$ are executed before any node in cluster $C_j$. Concretely, $C_i \prec_C C_j$ if $C_j$ is data-dependent on $C_i$, $C_j \overset{c}{\to} C_i$ (i.e., a node $k_j \in C_j$ is data-dependent on some node $k_i \in C_i$); or the two clusters are independent, $C_i \overset{c}{\not\to} C_j$ and $C_j \overset{c}{\not\to} C_i$. A set of clusters $\bigcup_{i=1}^{m} C_i$ defines a *valid partition* iff $\bigcup_{i=1}^{m} C_i$ defines a partition over $\mathcal{T}$, and if $C_i \prec_C C_j$, then $C_i \overset{c}{\not\to} C_j$.

*Cluster tiling.* In this phase, we consider each cluster independently and tile the nodes within a cluster into sub-kernels. For an arbitrary cluster $C$, its *tiling sequence* is a totally ordered set of all the sub-kernels generated within $C$ under the relation $\prec_{sch}^{C}$. Let $\delta_v^i$ denote the $i$-th sub-kernel of node $v \in C$ and $S_v$ be the set of sub-kernels inducing a partition on the blocks processed by kernel $v$ (Section III). Therefore, a tiling sequence for cluster $C$ is a totally ordered set on the set of sub-kernels $\mathcal{G}_C = \bigcup_{v \in C} \bigcup_{i \in S_v} \delta_v^i$. For any two arbitrary sub-kernels $\delta_u^i, \delta_v^j \in \mathcal{G}_C$, the tiling sequence must respect data dependency. Therefore, if $\delta_u^i \prec_{sch}^{C} \delta_v^j$, then $\delta_u^i \overset{k}{\not\to} \delta_v^j$.

*Combining the two phases.* As discussed above, our scheduling algorithm first finds a totally ordered set of clusters under the relation $\prec_C$, and then generates a tiling sequence for each cluster as defined by the order $\prec_{sch}^{C}$ on sub-kernels within a cluster. Thus, the final schedule is a totally ordered set under $\prec_{sch}$, where the relation is straightforwardly obtained via combining $\prec_C$ and $\prec_{sch}^{C}$. Intuitively, the objective of phase one is to provide a valid partition with the minimum cost, where the cost is the overall execution time of all tiling sequences. The objective of the second phase, which is executed interleaved with the first phase, is to find the best tiling sequence for each cluster, i.e., one which minimizes the overall execution time of all sub-kernels in the cluster. Below, we present the heuristic that implements the two phases.

*2) The Application Tiling Algorithm:* Every step of the iterative algorithm (Algorithm 1) involves both node partitioning and cluster tiling. Initially, each node is assigned to a unique cluster. The clusters are gradually enlarged by merging with their adjacent clusters. In every iteration, merging occurs between the two clusters with the highest weight on their linking edge. As mentioned, the weight reflects the maximum amount of performance gain achieved by merging the two border nodes of the clusters. After merging, the resulting partition replaces the old one only if it reduces the cost. To compute the cost of a cluster, we first need to tile it. The cost is then obtained by summing up the estimated execution times of all member sub-kernels (using the performance tables).

Our proposed tiling algorithm (Algorithm 2) aims to maximize the GPU utilization while maintaining a high cache performance. This is achieved by maximizing sub-kernel sizes subject to a cache-performance constraint. An exact cache analysis approach is not an efficient alternative. Moreover, it requires knowing the detailed cache configuration, which is not publicly available. Instead, we use the block memory lines, as provided by the block analyzer (see Section IV-B), to compute the memory footprint of a set of blocks. A constraint then ensures that the obtained memory footprint is not larger than the L2 cache size. We argue that using the memory footprint as an indicator for cache performance is a viable choice: A single block of *contiguous* memory addresses can fully reside in the cache if its footprint is not larger than the cache size. However, even if there are discontiguities in accesses, cache conflicts are largely avoided if the number of discontiguities is less than the associativity level of the cache, supposing that a memory block is larger than one cache way.

*Implementation of the application tiling algorithm.* As input, Algorithm 1 takes the application graph `appGraph` (Section III), the default execution time of each kernel `kerExeTimes` (Section IV-A), the edge weights `weights`, and a predefined weight threshold `thld` (Section IV-C). Other in-

**Algorithm 1:** Application Tiling heuristic

```
input : appGraph, kerExeTimes, weights, thld, inputs used by ClusterTile
output: Sch
1  foreach v_i ∈ appGraph do
2  |   C_i ← {v_i}; T_i ← {kernel(v_i)};
3  |   CO_i ← kerExeTimes[v_i];
4  |   push C_i to P;
5  end
6  candidEdges ← Select(weights, thld);
7  SortDesc(candidEdges); eix ← 0;
8  while eix ≠ end of candidEdges do
9  |   e ← candidEdges[eix];
10 |   C_a ← Cluster(src(e)); C_b ← Cluster(dest(e));
11 |   C_m ← MergeOrder(C_a, C_b);
12 |   P_tmp ← (P ∪ {C_m}) − {C_a, C_b};
13 |   if P_tmp induces a valid partition on appGraph then
14 |   |   T_m, CO_m ← ClusterTile(C_m, ...);
15 |   |   if CO_m < CO_a + CO_b then P ← P_tmp;
16 |   |   remove e from candidEdges; eix ← 0;
17 |   else
18 |   |   eix ← eix + 1;
19 |   end
20 end
21 foreach C_i ∈ P do
22 |   Sch ← Sch ∪ T_i
23 end
24 return Sch;
```

puts used by the cluster tiling algorithm (Algorithm 2) include the block dependency graph `blkDepGraph`, the block memory lines `blkMemLines` (Section IV-B), and the performance tables `perfTables` (Section IV-C). Initially, we assign each node to a unique cluster and set the cost of each cluster to the default execution time of its kernel (lines 1–5). We pick a set of candidate edges whose weights exceed the threshold, sort them in descending order of the weight, and set an edge index to point at the highest-weight edge (line 6–7). In each iteration of the while loop, we select the candidate edge at the index (line 9). Then, we discover the respective clusters containing the two ends of this edge (line 10) and merge these clusters (line 11). If the new set of clusters induces a valid partition (Section IV-C1) on the application graph (lines 12–13), then we tile the merged cluster using Algorithm 2 (line 14), redefine a new partition if the merged cluster reduces the cost (line 15), remove the edge from the set of candidates, and set the index to point at the next highest-weight edge (line 16). Otherwise, if the resulting partition is not valid, we try the next edge without removing the current one (line 18). The process continues until the set of candidate edges is emptied or no more valid partition can be created (line 8). Finally, the schedule is generated using the tilings of the final set of clusters (line 22).

*Implementation of the cluster tiling algorithm.* Given a cluster, Algorithm 2 provides the tiling sequence and its cost. We iteratively assign the blocks to sub-kernels until all blocks are assigned (line 3). Each iteration involves two rounds: *(i)* bottom-up round (lines 4–10), which prepares the minimum dependency requirements for processing block(s) of the leaf node(s) (there can be more than one leaf node in the cluster), and *(ii)* top-down round (lines 11–12), which tries to increase data-reuse efficiency and maximize utilization. To better understand these rounds, consider the kernels in Figure 1. During the bottom-up round, block $(0,0)$ of kernel B and blocks $(0,0)\ldots(3,0)$ of kernel A are selected to be assigned to sub-kernels $\delta^1_{\mathtt{B}}$ and $\delta^1_{\mathtt{A}}$, respectively. Subsequently, during the top-down round, blocks $(0,1)\ldots(0,3)$ of kernel B are also added to sub-kernel $\delta^1_{\mathtt{B}}$. This is because the dependencies of

**Algorithm 2:** ClusterTile heuristic

```
input : C_i = {c_1, ..., c_z}, blkDepGraph, blkMemLines, perfTables
output: T_i, CO_i
1  CO_i ← 0;
2  T_i ← ∅; assigned ← ∅; newSubKBlks ← ∅;
3  while assigned ≠ allClusterBlks do
   |   /* Bottom-up round                                              */
4  |   b_v ← Select the next unassigned block(s) from bottom kernel(s);
5  |   push b_v to toBeAssigned[v];
6  |   depList ← FindAllDeps(b_vj, blkDepGraph);  // direct and indir. deps.
7  |   while depList not empty do
8  |   |   b_v ← pop from depList;
9  |   |   if b_v ∉ assigned then push b_v to toBeAssigned[v];
10 |   end
   |   /* Top-down round                                               */
11 |   readyBlocks ←
   |     FindMoreBlks(toBeAssigned ∪ assigned, blkDepGraph);
12 |   foreach b_v ∈ readyBlocks do push b_v to toBeAssigned[v];
13 |   succeed ← CheckCacheConst(toBeAssigned, blkMemLines);
14 |   if succeed then
15 |   |   foreach b_v ∈ toBeAssigned[v] do  push b_v to newSubKBlks[v];
16 |   else
17 |   |   foreach c_v ∈ C_i do
18 |   |   |   subKer_v ← CreateSubKernel(newSubKBlks[v]);
19 |   |   |   CO_i ← CO_i + ET(subKer_v, C_i, perfTables);
20 |   |   |   add newSubKBlks[v] to assigned[v];
21 |   |   |   add subKer_v to T_i;
22 |   |   end
23 |   |   if T_i has not changed then return CO_i ← inf;
24 |   |   empty toBeAssigned & newSubKBlks
25 |   end
26 end
27 return T_i, CO_i;
```

these blocks are already covered by the dependencies of block $(0,0)$ of kernel B (as shown in Figure 1(b)). As a result, such dependencies can be met using the cache lines holding the corresponding data. Line 13 checks if the memory footprints of the sub-kernel blocks (which are to be assigned) fulfill the cache-size constraint. If the check succeeds, for each cluster node $\mathtt{c_v}$, we add all the newly found blocks to its set of new sub-kernel blocks `newSubKBlks[v]` (line 15). Note that each element of `newSubKBlks` is a set and items are added only if they do not already exist in the set. If the check fails, it means that the sub-kernels cannot be enlarged any more, without disturbing the cache performance. At this point, we compose the new sub-kernels out of the blocks that were iteratively added to `newSubKBlks` (line 18). In lines 19–21, we acquire the estimated execution time of the sub-kernel (by using the cluster to find the correct performance table and the grid size as the lookup index), update the cluster cost, mark the blocks as assigned, and add the sub-kernels to the tiling sequence. If, however, the algorithm does not manage to add any new sub-kernel, it returns the maximum cost indicating that the cluster cannot be tiled (line 23). Finally, `toBeAssigned` and `newSubKBlks` sets are emptied to collect the remaining blocks, which will be assigned to the next sub-kernels (line 24).

## V. Experimental Results

We use an Acer Aspire V15 laptop with NVIDIA GeForce GTX 960M dedicated graphics card as our experimental platform. The GPU has five Maxwell architecture streaming multiprocessors with the total of 640 CUDA cores as well as a 2 GB GDDR5 dedicated memory and a 2 MB L2 cache.

As the test case, we select an image processing application from the CUDA SDK. The application, called HSOpticalFlow, provides a GPU-accelerated implementation of the Horn–Schunck method [11] for optical flow estimation between

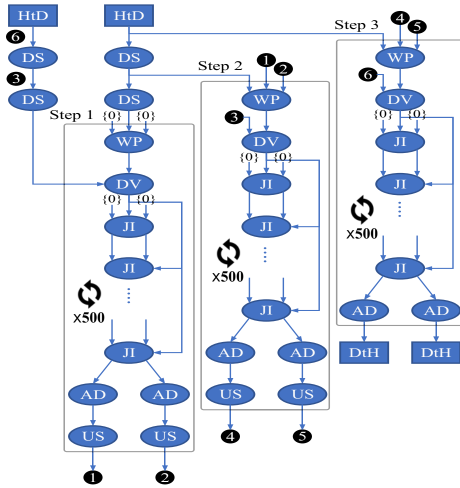


Fig. 4. DFG of the HSOpticalFlow application ({0} denotes a vector of zeros)

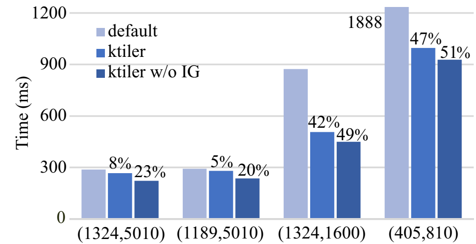


Fig. 5. Impact of KTILER on overall execution time

two frames. Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera [11]. The application is composed of several major steps that run in succession. Each step deals with a frame size that is equal to or is a fraction of the original input frame size. For this experiment, we use three major steps and two frames of 1024 by 1024 pixels. Figure 4 shows the application graph and the three steps (shown in rectangles). The first, second, and third steps deal with frames of 256, 512, and 1024 pixels on each dimension, respectively. The number of JI nodes per step is set to the default value of 500. A JI node implements one iteration of the Jacobi method to solve the corresponding linear equations. As JI nodes make up 98.5% of the total execution time, and also as they are more cache sensitive, we perform the tiling on these nodes.

We evaluate the effectiveness of KTILER under various GPU/memory frequency configurations. For each configuration, we measure the execution time of the application in three modes: First, we run the application in the default mode. Second, we run it according to the schedule produced by KTILER. We obtain the execution time by taking the arithmetic mean over 5000 runs. Third, we hypothetically assume that the overhead of the inter-launch gap (IG), which is the idle time between consecutive kernel launches, is zero. The purpose of this mode (denoted as KTILER w/o IG in Figure 5) is to evaluate KTILER based on the amount of time that is spent on actual processing of the data. Moreover, as discussed in Section II, the length of the IG can be reduced; for example, by optimizing the device driver or by using software techniques involving CUDA streams. To measure and then exclude the IG, we use the NVIDIA Timeline View tool.

Figure 5 shows the results for various GPU/memory frequency configurations. The percentage values over the bars indicate the achieved performance gains of each KTILER mode (i.e., with or without the IG) with respect to the default mode. On average over the four configurations, KTILER improves the performance by 25% with the IG (the middle bars in Figure 5), and by 36% without it (the right bars).

We would also like to discuss the following observations: First, regardless of the IG overhead, the performance gains of the two configurations with lower memory frequencies (i.e., 810 and 1600 MHz) are larger than the other two. This is because by lowering the memory frequency, the performance bottleneck moves further towards memory-related stalls and hence higher cache performance results in larger gains. Second, the IG has a larger impact on performance when the memory frequency is higher. The reason is that, as compared to the kernel execution time, the length of the gap is less dependent on frequency and thus the total gap length is roughly the same for different configurations. Therefore, at larger frequencies, where the kernel execution time is low, the IG contributes more to the overall execution time and the gain difference between the two KTILER modes is larger.

We mention that on our experimental platform, it takes KTILER approximately twenty minutes to generate the schedule for the given application and input set.

## VI. CONCLUSION

We presented a novel software approach to accelerate execution of GPU-based applications by exploiting the GPU L2 cache as a communication means between the processing nodes, instead of using the slow global memory. We also identified the features that must be present in a kernel in order to benefit from tiling.

## REFERENCES

[1] J. S. Meena *et al.*, "Overview of Emerging Nonvolatile Memory Technologies," vol. 9, 2014.
[2] Y. Xie, "Future Memory and Interconnect Technologies," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.
[3] S. Yu and P. Y. Chen, "Emerging Memory Technologies: Recent Trends and Prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 3 2016.
[4] F. N. Iandola *et al.*, "Communication-Minimizing 2D Convolution in GPU Registers," in *IEEE International Conference on Image Processing*, 2013.
[5] C. Li *et al.*, "Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.
[6] P. S. Rawat *et al.*, "Resource Conscious Reuse-Driven Tiling for GPUs," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.
[7] F. Bistaffa *et al.*, "Optimising Memory Management for Belief Propagation in Junction Trees Using GPGPUs," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.
[8] Y. Liang and X. Li, "Efficient Kernel Management on GPUs," *ACM Transactions on Embedded Computing Systems*, vol. 16, 2017.
[9] "Technical Report: Cache-Aware Kernel Tiling, An Approach for System-Level Performance Optimization of GPU-Based Applications," https://bit.ly/2NUJ0sO.
[10] M. Stephenson *et al.*, "Flexible Software Profiling of GPU Architectures," in *Proceedings of The Annual International Symposium on Computer Architecture*, 2015.
[11] B. K. P. Horn and B. G. Schunck, "Determining Optical Flow," *Artificial Intelligence*, 1981.