2016 IEEE 28th International Symposium on Computer Architecture and High Performance Computing

Partitioning GPUs for Improved Scalability

Johan Janzén, David Black-Schaffer, Andra Hugo
Uppsala University, Department of Information Technology
Uppsala, Sweden
{ johan.janzen, david.black-schaffer, andra.hugo }@it.uu.se

***Abstract*—To port applications to GPUs, developers need to express computational tasks as highly parallel executions with tens of thousands of threads to fill the GPU's compute resources. However, while this will fill the GPU's resources, it does not necessarily deliver the best efficiency, as the task may scale poorly when run with sufficient parallelism to fill the GPU.**

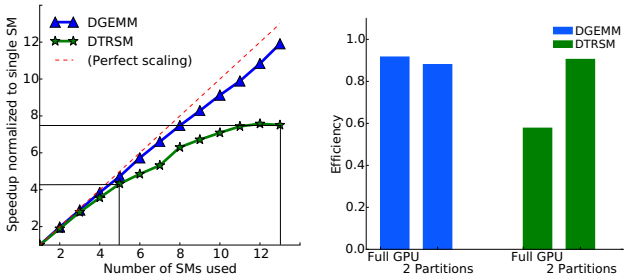
In this work we investigate how we can improve throughput by *co-scheduling* poorly-scaling tasks on sub-partitions of the GPU to increase utilization efficiency. We first investigate the scalability of typical HPC tasks on GPUs, and then use this insight to improve throughput by extending the StarPU framework to co-schedule tasks on the GPU. We demonstrate that co-scheduling poorly-scaling GPU tasks accelerates the execution of the critical tasks of a Cholesky Factorization and improves the overall performance of the application by 9% across a wide range of block sizes.

I. Introduction

GPUs have long been of great interest to the HPC community for their substantial raw computing power. To take advantage of this potential, however, applications must scale up to tens-of-thousands of threads on each GPU. Unfortunately, many of the key computational kernels in HPC applications lack this degree of scalability on today's GPUs. For example, adaptive mesh refinement is limited by intensive frontier communication [11], [16], sparse linear algebra solvers are limited by load imbalance across the dense submatricies [4], and fast multipole methods suffer from irregular control flow [3]. This limited scalability is visible as a decrease in *kernel efficiency* (performance per GPU resource, or, more specifically, performance per SM) as the kernel uses more and more of the GPU resources. In this work we investigate how we can *partition* the GPU to allow multiple kernels to *co-execute* with fewer GPU resources per kernel, and therefore higher *kernel efficiency* and higher overall throughput.

Applications targeting GPUs, and, in particular, GPU/CPU heterogeneous systems, are typically written in a task-based manner, which allows a runtime system to handle device and task scheduling. Traditionally, programmers provide the runtime with as large tasks as possible for it to offload to the GPU, in the belief that the larger the task, the better it will perform on the GPU. While large tasks often deliver better performance (throughput), there are two cases where this may not be the case: **1. Limited GPU scalability:** tasks that do not scale across the full GPU. In these cases, the kernel efficiency (performance per Streaming Multiprocessor (SM)) would be higher if the task could be scheduled on a subset of the GPU's SMs, and the overall performance could then benefit from scheduling other tasks on the remaining SMs. And, **2. Limited task-level parallelism:** applications whose task-level parallelism suffers from only issuing or executing a few large tasks, for example, due to precedence-constrained tasks that limit further parallelism. In these cases, the choice of larger sizes for GPU tasks and executing a single task on the GPU at a time can block the critical path through the applications' task graph and hurt performance.

In this paper we address both the GPU-scalability and the task-level parallelism problems by enabling controlled co-execution of tasks on the GPU. Through partitioning we force GPU tasks to run on fewer SMs, which improves the kernel efficiency (per-SM performance) of tasks with limited scalability. In addition, a partitioned GPU allows multiple tasks to execute concurrently, which can increase the available parallelism in precedence-constrained applications.



(a) (b)

Fig. 1: 1(a): Speedup normalized to using a single SM on an NVIDIA K20 GPU.
1(b): Efficiency (speedup divided by the number of SMs used) of `DTRSM` and `DGEMM` when executed on the full GPU (13 SMs, left) and two partitions (6 and 7 SMs, right).

The significance of limited GPU scaling for today's HPC applications is shown in Figure 1(a), where we compare the scalability of `DGEMM` and `DTRSM` from the MAGMA library on an NVIDIA Kepler K20 GPU. We see that `DTRSM` (green line) scales well up to 5 SMs, and thereafter the throughput gain per SM decreases. For the full GPU, the kernel efficiency (speedup over allocated SMs) is only 58%: a 7.5x speedup with 13 SMs. However, when run on only 5 SMs, `DTRSM` has a kernel efficiency of 87%: a 4.3x speedup with 5 SMs. In contrast, `DGEMM` (blue line) scales nearly perfectly, with a 92% efficiency when using the full GPU's 13 SMs, and only slightly better efficiency with fewer. The difference in scalability for

978-1-5090-6108-2/16 $31.00 © 2016 IEEE
DOI 10.1109/SBAC-PAD.2016.14

these two tasks indicates that there is a potential to improve overall application throughput if we can co-schedule tasks such that DTRSM operates in its most efficient region.

We can take advantage of the fact that DTRSM executes maximally efficiently on a subset of the GPU's total SMs by *partitioning* the GPU and running it on a subset of the SMs. Figure 1(b) shows the efficiency of executing DGEMM or DTRSM kernels, first with the kernels scaled to the full GPU and then with the GPU partitioned and two instances of each kernel co-executing. We see that executing two concurrent instances of DGEMM (left blue bar), which scales well on the full GPU, results in 2.5% lower efficiency compared to running on the full GPU (right blue bar). DTRSM (green bars), however, sees a dramatic improvement in efficiency when co-executing two instances on half of the SMs each, and is now able to efficiently utilize the full GPU, delivering in a 1.57x increase in throughput.

In order to co-execute kernels in this controlled manner, we use a software approach to partition commodity GPUs, similar to [17] (see Section VI on related work). With this approach, we can dynamically control the subset of SMs on which a task executes by adjusting how the kernel's grid is mapped to the SMs. We then enhance the StarPU runtime to be able to dynamically schedule tasks to the GPU partitions as well as to the available CPU cores, using its built-in performance-based scheduler. The scheduling strategy is therefore independent from the partitioning. This allows us to choose the partitioning and scheduling of GPU tasks depending on the structure of the application and the scalability of each task type. To better understand these tradeoffs, we provide a detailed exploration of task scalability and different partitioning configurations.

We evaluate our technique with a large, multi-kernel application: Cholesky factorization. By using a real application, with realistic dependencies and kernel diversity, we can provide a more realistic analysis of kernel co-execution than previous studies, which only analyzed pairs of kernels. Cholesky factorization is of particular interest as the expansion of its task graph is based on precedence-constrained DTRSM tasks, which we have seen suffers from limited scalability on today's GPUs. In this case the completion of the DTRSM task triggers the execution of highly-scalable DGEMM tasks. As a result, the application has the potential to benefit from increased efficiency by co-executing the limited-scalability DTRSM task on smaller partitions, if it does not suffer from running the highly-scalable DGEMM task on a smaller partition. To demonstrate this, we use a prototype implementation that modifies the GPU kernel code for execution, although these transformations could be readily automated in the compiler or as part of the GPU runtime.

This paper makes the following contributions:

- A kernel-based resource allocation implementation that enables partitioned GPU kernel execution within the StarPU framework for NVIDIA GPUs,
- An analysis of the scalability of key HPC GPU kernels,
- A runtime performance-model based solution for efficient GPU partitioning and scheduling, and,
- A demonstration of the combined approach with Cholesky Factorization that shows a 9% throughput improvement.

II. Partitioning Kernels on NVIDIA GPUs

To enable co-execution of kernels on NVIDIA GPUs, we need to overcome the limitations of the native runtime's scheduler. Our approach is based on a software technique that provides isolated co-execution by adjusting how a kernel's grid is mapped to the available SMs.

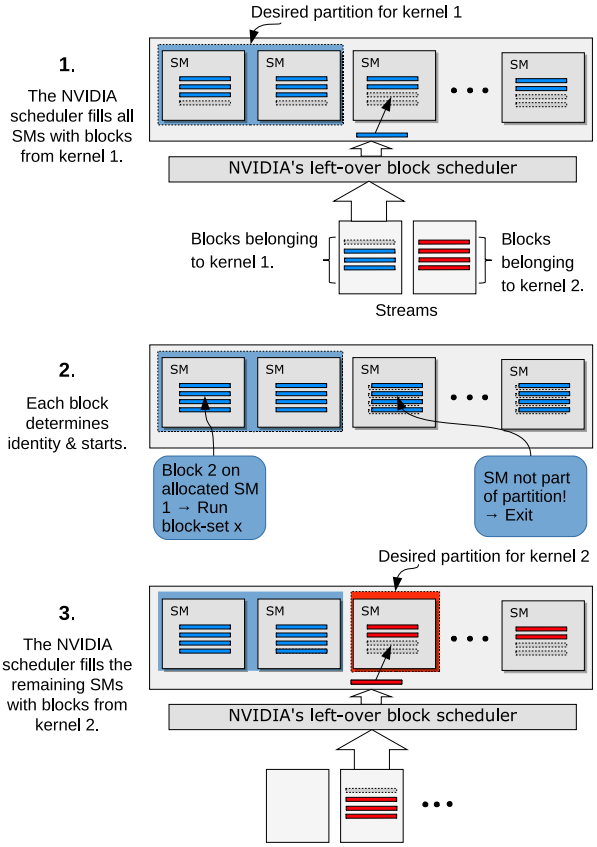


Fig. 2: Step-by-step description of how two kernels are launched into different partitions.

A. Current GPU Scheduling Capabilities

The standard approach for co-executing kernels on NVIDIA's Kepler architecture is through the use of different *streams* for independent kernels, typically referred to as *multi-streaming*. However, current NVIDIA drivers use a co-scheduling policy known as "left-over" scheduling [10], which only co-executes kernels if (or when) they do not fill the GPU. As most kernels are launched with far more blocks than the hardware can support at once, this effectively results in serial execution except at the very end of a kernel when it has too few warps remaining to fill the GPU. This scheme avoids co-execution in most cases, and makes it difficult to analyze and predict kernel performance, as the actual start of the execution is highly dependent on what is happening in the other streams. The NVIDIA Grid Management Unit (GMU)

further complicates performance analysis as it allocates kernel blocks to SMs in a breadth-first fashion, which causes co-executing kernels to share and contend for compute resources within the SMs.

B. Software Partitioning of GPUs

In order to overcome the limitations of today's "left-over" schedulers, we devise a block-to-SM re-mapping technique to isolate kernels to particular sets of SMs, and thus effectively partition the GPU. By doing so we avoid sharing compute resources between kernels, as each SM only executes blocks from the same kernel. This allows us to remove much of the unpredictability in the co-execution performance[1]. With this technique we can specify the number of SMs for each kernel, which allows us to quantitatively measure the scalability of kernels on the GPU (Figure 1(a)), and provide this information to the StarPU runtime scheduler's performance models.

The key to our co-execution approach is that if we limit a kernel to only execute on a subset of the SMs, there will always be a "left-over" set of SMs available. As a result, the NVIDIA scheduler will now co-execute a subsequently launched kernel on this "left-over" set. In this manner we can force kernels to truly co-execute and control what fraction of the GPU's SMs each receives. This allows the StarPU runtime scheduler to view each GPU partition as an independent computational unit, and more accurately predict kernel performance (turnaround times for tasks) as kernels are effectively running on their own, smaller, but independent, GPU.

To achieve partitioned execution, we use the knowledge of the block scheduling behavior to remap a kernel's blocks to only the SMs upon which it should run. This source-to-source modification consists of 2 steps:

1. Mapping control. Before the launch of a kernel, the host first calculates the number of thread blocks required to exactly fill a single SM for that kernel, m (Listing 1), and the number of blocks needed to fill all SMs on the GPU, b.

We then build a $mapping$ in the GPU global memory that specifies how many blocks from this kernel should reside on each SM. For the SMs we want to be a part of the partition, we set the value m to fill that SM with the kernel's blocks, while we set all the value to 0 for all other SMs.

Finally we launch the kernel with just enough blocks (b) to fill all SMs on the GPU, adding the $mapping$ and the original grid dimensions to the list parameters of the kernel function.

On the device-side, we insert control code at the start of each kernel's execution that reads the ID of SM upon which it resides, and uses that value to atomically read and decrement the corresponding entry in the global $mapping$. If the value is less than 1, then the block should not do any work, and exits. Otherwise the block uses this value, together with its SM ID and the mapping, to calculate which work it should do.

After this short setup sequence, the kernel code on the unmapped SMs has exited, leaving them free for further scheduling, and the mapped SMs are filled with blocks from the kernel. As the kernel was launched with only as many blocks as can fit all the SMs, there are no further blocks from this kernel to schedule.

2. Block wrapping. Since we have launched the kernel with a different number of blocks from what it was originally designed for, each physical block must now do more work. A solution to this problem was proposed by Stratton *et al.*[12], wherein they executed CUDA kernels on multicore CPUs with a technique they call *iterative wrapping*, which lets a few threads iteratively do the work of many. This approach has also been extended for non-partitioned (sharing SMs), co-execution of GPU kernels by Pai *et al.* [10]. We extend this technique to *partitioned* co-execution by letting each block do the work of zero, one, or more of the original blocks. The work allocation is determined by spreading the original grid dimensions (original number of blocks) across the scheduled blocks in a modulo fashion.

Each scheduled block executes its assigned work with a loop that surrounds the original kernel logic. Inside the kernel logic, we replace all references to the CUDA's blockIdx and grid dimension references with ones derived from the loop index and original dimension. We do not change *threadIdx* references, as the size of the blocks remain unchanged.

As a result, each block determines if it should run on a given SM, and, if so, what work it should do. The block then executes this work as the original kernel would, but instead of exiting at the end, it then looks to see if it has further work. As a result we can control what portion of the SMs are allocated to each kernel and enforce partitioned co-execution by preventing different kernel's blocks from sharing the same SM.

```
m = getMaxBlocksPerSM(kernel,threads)

// Execute on SM 0 and 1 only
int mapping[SMS_ON_THE_DEVICE] = {m,m,0}

b = SMS_ON_THE_DEVICE*m
kernel<<<b,threads,0,stream>>>(mapping,originalGrid, args...)
```

Listing 1: Pseudo code snippet for launching a partitioned kernel, host side

```
smid = getSMid()
nb = readAndDec_OncePerBlock(mapping[smid])
if(nb <= 0) return  // Exit if we should not run on this SM

// Calculate a global identifier for each block
i = f(smid,nb,firstMappedSMid)

// For each of the SM's blocks call the kernel code
for(block : BlocksToExecute(i)) work(block,gridDim)
```

Listing 2: Pseudo code snippet for executing a partitioned kernel, device side

C. Using multiple partitions

For multiple partitions we assign a CUDA stream to each partition and take advantage of the "left-over" scheduling policy to place our modified kernels on the correct SMs. Figure 2 shows this procedure for 2 partitions. The GPU schedules kernels from streams round-robin at a kernel granularity. This means that it always launches all blocks from a single kernel before switching to the next one. At step 1 the scheduler fills

[1]The major remaining shared resources are the global memory system: L2 and DRAM.

up the entire GPU with blocks from kernel 1. As the blocks that end up on a SM where they should not be exit rapidly (step 2), the GPU can immediately start launching blocks from the second kernel (step 3). A few blocks from the second kernel will remain in the queue after filling the GPU in this case, because the device was not empty. These blocks drain rapidly through the first available SM and have minimal impact on performance. This process can be repeated as desired until all SMs are part of a co-executing partition.

Since each SM is dedicated to a single kernel, kernels do not contend for SM-resources, which prevents them from impacting each other's performance. This characteristic is essential for our scheduler to be able to learn and predict kernel execution times independently from other co-executing kernels.

III. Dynamic Scheduling on CPUs and GPU Partitions

Our approach allows us to reliably measure the execution time of a GPU kernel on a specified subset of SMs as kernels are isolated to their own SMs. We can use this information to schedule tasks on either CPUs or partitions of the GPU to minimize overall execution time. To accomplish this, we integrate our GPU partitioning with the StarPU runtime system to take for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units.

A. Scheduling in StarPU

The core principles of StarPU [5] are that tasks can have several implementations (e.g., CPU and GPU) and that data transfers are handled transparently by the runtime system (e.g., CPU to GPU). To minimize data transfers, StarPU allows multiple copies to reside on different processing units as long as the data is not modified, and uses asynchronous data prefetching to hide latencies.

StarPU provides a flexible scheduling platform for strategies ranging from greedy and work-stealing based policies to more elaborate Minimum Completion Time (MCT) policies [14]. This latter family of schedulers builds history-based performance models for each task by observing the execution at runtime. As a result, the runtime can provide a relatively accurate estimation of each task's performance on each resource, which allows the scheduler to take appropriate decisions when assigning the tasks to a computing resource.

B. Co-scheduling Tasks on GPU

Co-executing multiple tasks on a GPU requires a CUDA stream for each modified partition and the ability to launch the tasks independently. While StarPU natively supports multiple CUDA streams, it needs to be extended to understand our GPU partitioning so that it can collect appropriate performance information for the kernels on a given partition. To do so, we extend StarPU's *scheduling context* structure [8], which encapsulates the notion of a parallel task and restricts its execution to a section of the machine (e.g. a subset of CPUs, or CPUs and GPUs).

We extend the *scheduling contexts* to group not only CPUs and GPUs but also a subset of a GPU's SMs. In order to ensure the co-execution of the kernels we map each scheduling context to a stream. Tasks assigned to a certain scheduling context are then launched to the corresponding stream. By using this structure we associate an SM allocation to a stream, which allows StarPU's scheduling strategies to consider how long a task would take on a certain partition.

Once the task is assigned to a *scheduling context*, the StarPU driver will move the data to the GPU (if needed) and launch the kernel. In order to avoid potential bottlenecks at the kernel launching time, we designate one core for each CUDA stream.

The optimal number of partitions is tied to the granularity and parallelism of the tasks. Our prototype requires the programmer to specify the subset of SMs for each partition and build the appropriate scheduling contexts at the beginning of the execution. Ideally, this parameter would be chosen automatically by exploring different partitioning configurations at runtime.

C. Performance Models for Partitioned GPUs

In order to efficiently schedule tasks, we need to be able to estimate the time they will take to execute on the GPU partitions and on the CPU cores. We rely on StarPU's built-in profiling history table, which is updated dynamically as soon as tasks execute on different computing resources. We extend this facility to store the execution time of the tasks separately for different GPU partition sizes (from 1 to 13 SMs). it itThis information allows the scheduler to predict the execution time of a task on any of the available partitions and the CPU cores. As a result, the scheduler is independent of the partitioning and will automatically adapt to different partitioning configurations.

D. Minimum Completion Time Scheduler

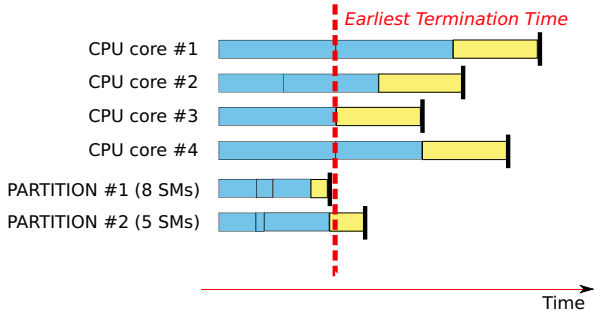


Fig. 3: MCT scheduling policy: blue rectangles are execution times of tasks already assigned, the yellow rectangle is the estimated execution time on each computing resource of the task to be scheduled

We extend StarPU's Minimum Completion Time scheduler to place tasks across the CPU cores and GPU scheduling contexts (partitions) based on the estimated execution time of the tasks. This strategy (see Figure 3) relies on the history-based

performance model to predict how long a task would require to execute on each computing resource (yellow rectangles) as well as to estimate the time required to finish executing the tasks already scheduled (blue rectangles). Based on this information it assigns a task to the computing resource on which it is expected to terminate the earliest (partition 1, in the figure). Depending on where its input data resides, the task's execution time may include the data transfer time.

IV. Application: Cholesky Factorization

To demonstrate the potential of our GPU co-execution and performance-model based scheduling strategy, we apply them to the Cholesky Factorization application from the dense linear algebra library Chameleon [13], [1], implemented on top of StarPU. We adapt the implementation of the kernels to execute on the restricted number of SMs as controlled by the runtime system (see section II). To accomplish this we use the open source MAGMA GPU kernels as they allow us to modify the kernel code for our GPU partitioning[2].

A. Cholesky Factorization on Top of MAGMA

The Cholesky Factorization consists of 4 types of tasks [2]: `DSYRK`, `DTRSM`, `DGEMM`, and `DPOTRF`. Each of these are available in the Intel MKL library (CPU), and from MAGMA (GPU), while only the first three are provided by cuBLAS. In order to co-execute on the GPU, our prototype implementation requires access to the kernel code to change how the grid is executed across the SMs. However, this is not possible with the cuBLAS kernels as the code is not available. For our evaluation we changed Chameleon to call MAGMA kernels, for which the code is available[3].

V. Evaluation

With our ability to control the SMs used by a kernel, we can first study the GPU scalability of each kernel. This data shows us that we can more efficiently use the GPU by assigning a reduced number of SMs to the tasks that do not scale to the entire device, while allowing different computations to co-execute on the remainder of the GPU. With this approach we can deliver a 9% performance improvement for Cholesky Factorization.

A. Experimental platform

All measurements are done on a 2-socket, 8-core Intel(R) Xeon(R) E5-2680 CPU with a base frequency of 2.70GHz and 64 GB of RAM, with an NVIDIA Tesla K20 graphic card. The K20 contains 13 Stream Multiprocessors (SMs), each with 192 "cores" and 5 GB of GDDR5 memory. We use Intel MKL 11.3.0, Chameleon 0.9.1, MAGMA 1.7.0, CUDA 7.5, and StarPU 1.2.0 on Linux 2.6.

The StarPU runtime system requires a core to act as driver for each GPU stream, which reduces the number of CPU cores available for scheduling CPU tasks by the number of GPU partitions we run. For each configurations we run the application 5 times to improve the precision of the performance model and then we average the performance of 5 executions.

[2]While our prototype requires manual code modifications, this could be readily automated in the compiler or runtime.

[3]cuBLAS is generally more efficient, and as a result Chameleon typically only uses MAGMA for `DPOTRF`. Cholesky Factorization with cuBLAS can reach up to 1100 GFLOPS compared to 800GFLOPS for MAGMA.

B. Study of Cholesky Factorization's tasks

The Cholesky Factorization application has data dependencies between tasks that make the `DTRSM` tasks one of the most critical. These dependencies are shown in Figure 4, where we can see that the `DTRSM` tasks (red) must be completed before a significant number of `DSYRK` and `DGEMM` (blue) tasks can begin execution. If the `DTRSM` tasks execute slowly, for example, due to poor scaling if assigned the whole GPU or executing serially on a single CPU core, this can hurt the overall performance of the factorization.

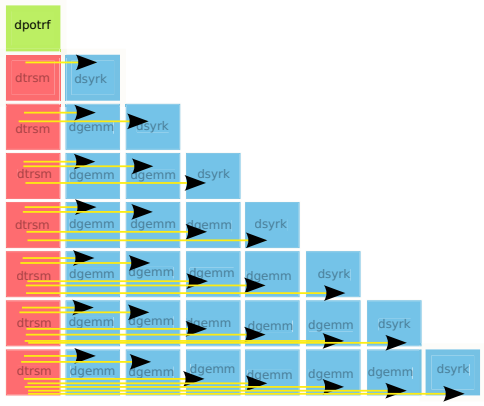


Fig. 4: Cholesky Factorization data flow for the first iteration on a matrix 8x8 blocks

To investigate the scalability of typical GPU tasks within Cholesky Factorization, we vary the SM allocation and measure the *throughput* of the kernels. In Figure 5 we see how the 4 types of tasks scale on different partitions as a function of the size of the input problem. We selected measurements for 3, 6, and 13 of the 13 total SMs on our GPU.

The data in Figure 5 shows that task scalability on the GPU varies with both the size of the problem and with the type of task. For `DGEMM`, scalability is good for even very small size problems, scaling up to the full 13 SMs at problem sizes as low as 1000. However, for `DTRSM`, problems below size 2000 exhibit poor scaling. Indeed, `DTRSM` achieves *twice the per-SM efficiency* at a problem size of 1500 when *running on half* of the GPU's SMs compared to running on the full device. This indicates that a scheduler that can find other work to co-execute on the GPU with `DTRSM` could improve *overall throughput* by reducing `DTRSM`'s GPU allocation when it executes.

C. Cholesky Factorization on a Partitioned GPU

In order to cope with the lack of scalability of the `DTRSM` tasks on the GPU we use our technique to co-execute several kernels in parallel, and thereby enable `DGEMM` and `DSYRK`

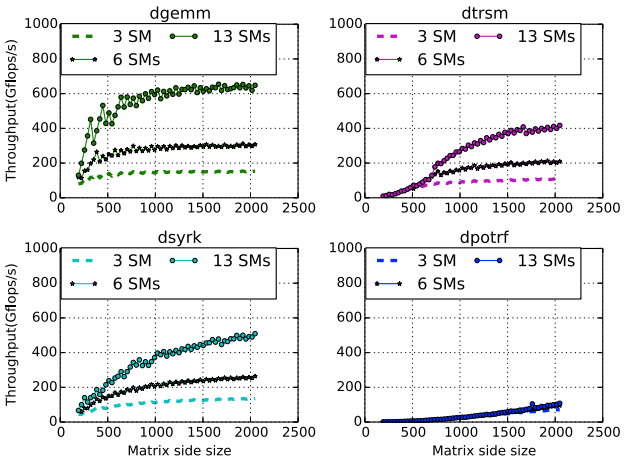


Fig. 5: Performance on different sizes of partitions for the 4 types of tasks of the Cholesky Factorization

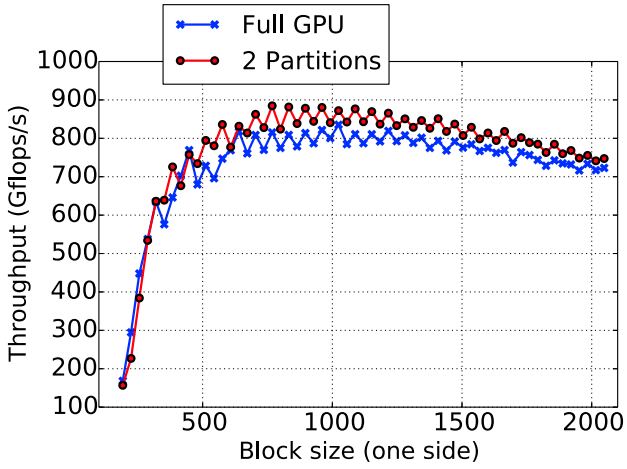


Fig. 6: Scheduling tasks on 15 CPUs + GPU (13 SMs) versus 14 CPUs + GPU (6 + 7 SM partitions)

to start sooner. Specifically, we partition the GPU in two (partitions of 6 and 7 SMs) and use the MCT scheduler to schedule the tasks composing Cholesky across these two GPU partitions and the remaining 14 CPU cores. We compare this approach to the one where we do not use any partitioning, and thus schedule tasks with the same MCT scheduler but on 15 CPU cores and the entire 13-SM GPU. We report the results from factorizing a *square matrix of 26880 x 26800* double precision elements and we vary the size of the blocks, from *192 to 2048*.

Figure 6 clearly shows the benefit of partitioning the GPU. The red line corresponding to the partitioned version significantly outperforms (by up to 9%) the non-partitioned (blue) version. By favoring the co-execution of the `DTRSM` tasks with other task of the same or different type we have more efficiently used the system's resources, and thereby improved the overall throughput.

These results are particularly good because through our approach we are able to improve the GPU utilization for a few poorly scalable tasks (the number of `DTRSM` tasks is of order n^2) and thus allow exposing sooner the high parallelism of the many `DGEMM` tasks (of order n^3) without an impact on their performance. For tasks with an input block size smaller that 320 our approach does not show a benefit due to the overhead.

D. Overhead

Our software partitioning incurs overhead from two sources: the host-side kernel launch setup (Listing 1), and the extra code added to the kernel itself (Listing 2).

As the four types of tasks of the Cholesky Factorization are implemented with a series of calls to variations of the `DGEMM` task and kernel, we limit the overhead analysis to the `DGEMM` task itself. We measured the kernel launch setup overhead to approximately 72μs. This overhead is significant when launching kernels smaller than 192. For larger task sizes the CPU-time completely overlaps GPU runtime.

The kernel modification includes two small overheads: one constant overhead from the mapping control (Section II) and per-block overhead from the loop executing the blocks. We measured the first overhead to be 1.1μs. The second overhead is more complex. As an example of this, we measured the partitioned `DGEMM` with a task size of 2048 across 100 runs. At this size the constant overhead is a negligible as the total kernel runtime is 27.5ms. We then repeated the measurement with an unmodified (unpartitioned) kernel and found the runtime was greater: 33ms. In this case, the modified kernel is actually faster than the original, possibly due to the reduced overhead of having a significantly smaller grid or caching effects due to a different GPU execution.

E. Impact of different GPU partitioning

The ideal number and size of the GPU partitions is dependent on the application. Our solution allows the programmer to choose the partitions, but then automatically schedules the tasks onto them.

In this section we sweep different configurations in order to see how sensitive Cholesky Factorization is to different partitionings. We first evaluate different distributions of SMs when we partition the GPU in two and then we extend the evaluation to more than two sub-sets of SMs.

According to Figure 5 if we want to partition the GPU in two, the best way would be to give at least 6 SMs to the `DGEMM` and `DSYRK` tasks for sizes larger than 500, without considering `DPOTRF`, as the performance model is likely to choose to execute it on the CPUs. On the other hand, `DTRSM` does not scale particularly well on 6 SMs for tasks smaller than 1300-1400, and thus assigning a task to such a partition would effectively waste 3 SMs. However, as the application progresses, more of the tasks will be `DGEMM`, and we would like a GPU partitioning that favors a fast execution for them. In our current prototype (which does not support dynamically changing the partitions) this becomes a trade-off between the

benefit of accelerating the execution of the DTRSM and an efficient use of the GPU for the later DGEMM tasks.

Our experiments show that creating an unbalanced partitioning when the application is regular, as it is the case of the Cholesky Factorization, may result in scheduling a critical task on a smaller partition and slowing down the critical path. Thus creating two partitions of 1 and 12 SMs actually slows down the overall execution by up to 5 %, as the tasks running on the 1 SM end up delaying the execution. However, more balanced configurations do not show a significant impact on the performance of the application (generally under 5% slowdown). Based on the performance model of the tasks on the two partitions and the CPU cores, the scheduler assigns the tasks to minimize the execution time.

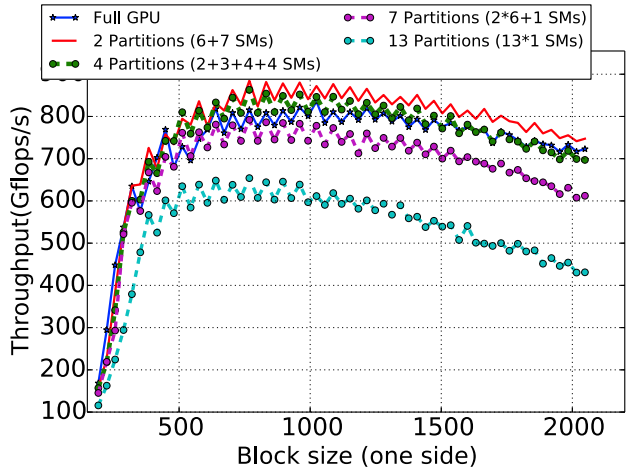


Fig. 7: Partitioning the GPU in different number of partitions

When partitioning the GPU into more than two partitions (see Figure 7) we observe that performance generally decreases with more partitions. We see up to 10% slowdown (for 7 partitions vs. two partitions) and between 20% and 40% (peak worst) for 13 partitions. For block sizes between 400 and 500, having four partitions may slightly improve the performance (5%). This is due to the fact the GPUs benefit from the co-execution of several small tasks that are not scalable on a large number of SMs. Larger tasks (especially DGEMM) are delayed by such a small set of computing resources. At this point the benefits of accelerating the DTRSM are overcome by the increased time DGEMM takes to execute before it frees dependencies for the next iteration of the factorization.

Such a fine-grain partitioning of the GPU should benefit very small tasks, however scheduling small tasks on the GPU is limited by both the overhead of the scheduling strategy (significant when the execution time of a task is small) and the serialized data transfer for the co-executing tasks on the GPU.

In the extreme, if we exclude this overhead our approach allows us to consider a single SM as an independent processing unit. If we look at dedicating a CPU core per SM to act as a driver instead of performing computation, we observe in Figure 8 that in most of the cases 1 SM outperforms a

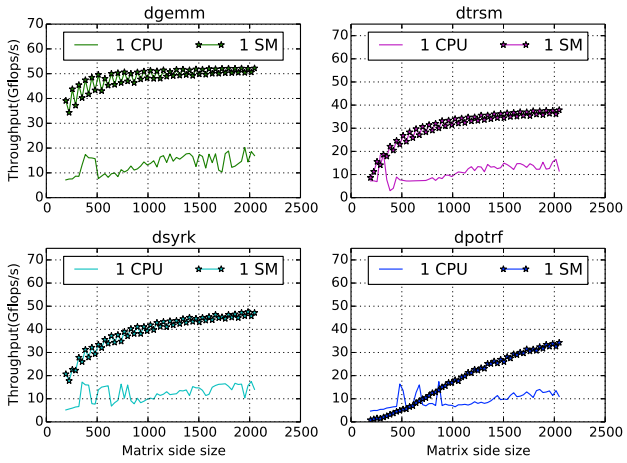


Fig. 8: 1 CPU core vs 1 SM

CPU core. Most of the tasks scale uniformly on 1 SM (with a throughput of up to *50 Gflops/s*) but highly irregular on the CPU due to caching effects. On the CPU, tasks reach the highest performance for a size of matrix around 500 (*17 Gflops/s* for DGEMM), while for higher values the performance decreases and then increases as the Intel MKL library blocks the larger computations into smaller ones [9].

VI. Related Work

Previous work proposed solutions for GPU co-execution at the hardware level [15][6] (evaluated through simulation) or at the software level by transforming the kernels. These proposals either merged co-executing kernels before launch [7] or adjusted the grid size and the block size of the kernels [10].

Awatramani *et al.* [6] investigated the impact on throughput of mixed and partitioned execution of pairs of kernel, through simulation. Their results showed that mixing bandwidth- and compute-bound blocks is often beneficial, unless one of the kernels has a high L1 miss rate. Although mixed co-execution performed better on average, in order to predict performance, one must be aware of the exact kernel pairing. Our work differs from this by co-executing two or more kernels with *partitioned* execution, and evaluates a full complex application executed on a commodity heterogeneous platform, rather than just individual kernels in simulation.

Ukidave *et al.* [15] explored partitioned execution of OpenCL kernels with adaptive partition sizes through modifications of the hardware block scheduler in the Multi2Sim simulator. Their scheduling strategies were based on an occupancy algorithm that maps workgroups to computational units based on whether the kernel requires a fixed amount of computational units or not. Our solution schedules the tasks on the computational unit that would allow the application to finish fastest, rather than looking at individual kernel partitioning, and targets a heterogeneous platform of both CPUs and GPU.

Gurevara *et al.* [7] proposed a software solution for concurrent kernel execution by merging pairs of kernels. They

found that small kernels that would not occupy the full GPU generally benefit from merging, however the combined resource usage may decrease SM occupancy and limit the gains. Additionally, merging kernels with unequal execution times penalizes the shorter-running kernel and occupies its resources for a longer period. Our work leverages existing hardware to achieve true, independent co-execution of kernels, and is not limited to pairs.

Pai *et al.* [10] proposed extending iterative wrapping [12], to decrease the parallelism of kernels to allow pairs to co-execute on the same SMs (non-partitioned). They statically reserved resources in the SM for the second kernel, but since kernels differ in resource utilization, they could not control the degree of concurrency, and resorted to an average allocation policy. Their evaluation used replays of CUDA API traces instead of full heterogeneous applications. Our work is similar in that we both extend Stratton *et al.*to control parallelism, but our iterative-wrapping extension is then combined with the block-to-SM mapping technique used to *isolate* kernels to particular sets of SMs. By doing so we largely avoid sharing compute resources, which prevents unpredictable co-execution performance, while giving us full control of the degree of co-execution and can support more than just pairs of applications.

Wu *et al.* [17] proposed a similar software solution for partitioning. Their approach has each SM fetch blocks at runtime from a centralized queue, rather than using a code transformation to locally compute the next block as we do. While obtaining the blocks centrally provides more flexible scheduling, the global load needed to do so prevents compiler optimizations as a new dependency is introduced. The impact of this is seen where Wu *et al.* report 2.5-6% overhead for `DGEMM`, while we see a small speedup. More importantly, we evaluate kernel co-execution on a full application with accurate dependencies and kernel diversity driven by a performance-aware scheduler, whereas they investigate arbitrary pairs of kernels.

VII. Conclusion

In this work we demonstrate the benefits of partitioning the GPU to allow tasks that do not scale well on the entire GPU to co-execute with other tasks for improved throughput. We show that this solution is particularly important for critical tasks where executing them serially across the full GPU limits throughput and available parallelism.

To accomplish this we developed a software technique to enable co-executing kernels on different partitions of current NVIDIA GPUs. We implemented a novel grid-to-SM mapping and integrate it into the StarPU task-based runtime system's scheduling strategies. As a result we can automatically schedule kernels across heterogeneous mixtures of CPUs and GPU partitions. We demonstrate the potential of this approach with the Cholesky Factorization, which shows a 9% throughput improvement across a wide range of block sizes.

To make this approach more generally applicable, we plan to extend our scheduler to dynamically adjust GPU partitioning at runtime and automate the kernel transformations in the compiler. These advances will allow us to adapt to different application behavior and eliminate the need for the programmer to specify the partitioning.

VIII. Acknowledgments

We would like to thank the Chameleon and MAGMA developers for assistance. This work was funded by the Swedish Foundation for Strategic Research under grant FFL12-0051.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," vol. Vol. 180, 2009.

[2] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs," in *GPU Computing Gems*, Wen-mei W. Hwu, Ed. Morgan Kaufmann, 09 2010, vol. 2.

[3] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, 2015.

[4] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Task-based multifrontal QR solver for GPU-accelerated multicore architectures," IRIT, Toulouse, Research Report IRI/RT–2015–02–FR-r1, Jun. 2015.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, Aug. 2009, pp. 863–874.

[6] M. Awatramani, J. Zambreno, and D. Rover, "Increasing gpu throughput using kernel interleaved thread block scheduling," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 503–506.

[7] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," in *Workshop on Programming Models for Emerging Architectures*, vol. 9. Citeseer, 2009.

[8] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," in *IPDPS Workshops*, 2013, pp. 1050–1059.

[9] Intel Corporation, "Optimize for Intel AVX Using Intel Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine," https://software.intel.com/en-us/articles/.

[10] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, p. 407, 2013.

[11] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie, "Efficient gpu-implementation of adaptive mesh refinement for the shallow-water equations," *Journal of Scientific Computing*, vol. 63, no. 1, pp. 23–48, 2015.

[12] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained spmd-threaded programs for multicore cpus," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. ACM, 2010, pp. 111–119.

[13] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1 –8.

[14] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[15] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 2014, pp. 168–175.

[16] P. Wang, T. Abel, and R. Kaehler, "Adaptive mesh fluid simulations on GPU," *New Astronomy*, vol. 15, no. 7, pp. 581 – 589, 2010.

[17] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. ACM, 2015, pp. 119–130.