On block-asynchronous execution on GPUs

Hartwig Anzt, Jack Dongarra
University of Tennessee
{hanzt,dongarra}@icl.utk.edu

Edmond Chow
Georgia Institute of Technology
echow@cc.gatech.edu

***Abstract*—This paper experimentally investigates how GPUs execute instructions when used for general purpose computing (GPGPU). We use a light-weight realizing a vector operation to analyze which vector entries are updated subsequently, and identify regions where parallel execution can be expected. The results help us to understand how GPUs operate, and map this operation mode to the mathematical concept of asynchronism. In particular it helps to understand the effects that can occur when implementing a fixed-point method using in-place updates on GPU hardware.**

***Keywords*-GPU-computing, asynchronous execution, block-asynchronous iteration**

I. Introduction

Understanding the operation mode of streaming processors like GPUs can be important for implementing linear algebra algorithms. In particular when implementing fixed-point iterations that reuse the input vector as output vector, understanding the order in which the distinct vector entries are updated is required as this order can have significant impact on the convergence properties of the iteration method [1]. On GPUs, the order in which the vector entries are updated is determined by the level of parallelism and the thread block scheduling.

In this paper we unveil some key properties of the operation mode used by GPUs. Precisely, we experimentally locate instances where vector values were updated subsequently, and try to identify the largest sub-vector for which parallel execution can be expected. The goal is to understand how GPUs execute vector instructions, and the impact on numerical algorithms that reuse the input vector as output. The most prominent representative for this class of algorithms are fixed point iterations that become "block-asynchronous [2]" when executed on GPUs.

The rest of the paper is structured as follows. In Section II we list some related work on asynchronous two-stage and block-asynchronous iterations on GPUs. In Section III we review some technical details about the hardware characteristics of a state-of-the-art GPU architecture, the CUDA programming model, and the kernel execution mode. The core of the paper is Section IV, where we employ a light-weight test kernel to experimentally analyze some properties of the kernel execution of the GPUs. We summarize the experimental observations in Section IV-C.

II. Related work

Relaxing the level of synchronicity in favor of more efficient execution on GPUs has been subject to numerous research efforts. Most notably is the idea to span the arc between asynchronous two-stage iterations and the block-asynchronous execution mode: a fixed-point iteration reuses the input vector as output vector; the block-asynchronous execution of the GPU destroys the global synchronicity; the result is an algorithm where parallel handling of subsets is embedded in a asynchronous global update scheme. For a Jacobi relaxation method this setting results in a "block-asynchronous Jacobi" where subsets of the iteration vector are iterated in synchronous Jacobi fashion, and asynchronous updates are used in-between the subsets [2]. The potential of block-asynchronous Jacobi on GPU hardware was initially investigated in [3]. This includes an analysis on the benefits of adding local iterations on shared memory that may be used to hide memory latency and promote faster solver convergence. Block asynchronous Jacobi was also considered as smoother for geometric multigrid methods [4], and evaluated in a mixed-precision iterative refinement framework [5]. In [6], a block-asynchronous Jacobi was used in a multigrid solver running on a GPU cluster. A Gauss-Seidel method as inner solver was considered in [7] where a block-relaxation method was used to solve a stencil discretization. In [8], block-asynchronous Jacobi was employed as an iterative solver for sparse triangular systems arising from incomplete factorization preconditioning. A fixed-point implementation allowing for block-asynchronous execution was also used in [9] for the iterative generation of an incomplete LU factorization. Some of these previously mentioned publications make vague or incorrect statements about the size of the subsets where the synchronous inner solver acts.

III. Technical Details

In the following paragraphs we review some technical details that provide an idea of what we can expect in the experimental analysis.

A. Hardware characteristics

The GPU hardware we target in this paper is NVIDIA's Kepler GK110 (GK110b) architecture [10]. It is the latest architecture featuring full double precision support and, e.g.,

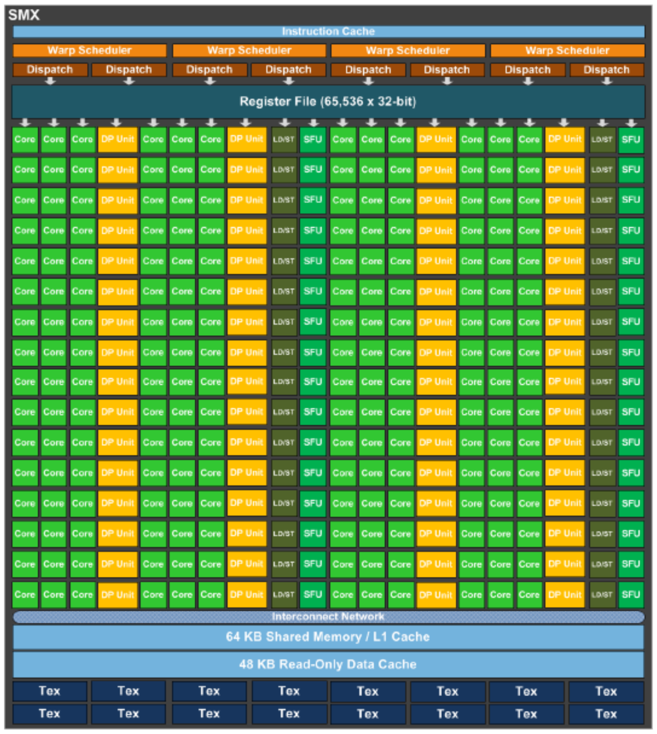

Figure 1: GK110 SMX design [10].

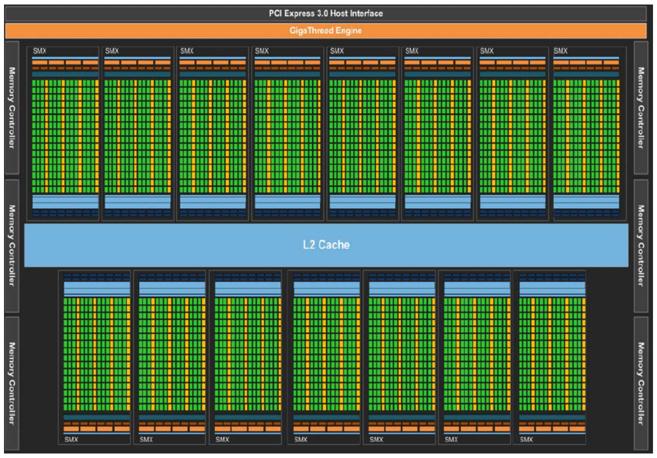

Figure 2: GK110 architecture [10].

used in the K40 GPU line. One K40 composes of 15 multiprocessors called “streaming multiprocessor (SMX),” each of the SMX equipped with 64 KB of local multiprocessor memory. The local memory splits into shared memory and L1 cache. On each multiprocessor, there are 192 single-precision CUDA cores, 64 double-precision arithmetic units, 32 special functional units, and 32 load/store units. The 32 load/store units handle the main memory access, which means that each memory transaction can read or write up to 128 byte of contiguous memory. Working with 32-bit data types, this means that each memory transaction can read or write up to 32 values, if these values fall within the same 128B segment of memory. Otherwise, part of the bandwidth is unexploited. The coalescing of up to 32 data elements into a single transaction is motivated by the size of the “warp.” The arrangement of the building blocks forming one SMX is sketched in Figure 1. 15 multiprocessors are aggregated in one K40 GPU, all accessing 12 GB of main memory at an accumulated bandwidth of 288 GB/s (theoretical peak). Additionally, each card is equipped with 1.5 GB of L2 cache, and a PCI controller managing the GPU-host communication. Figure 2 visualizes the GK110 architecture.

B. *CUDA programming*

CUDA C [11] is an extension to the C programming language that allows for general purpose computing on NVIDIA’s CUDA architecture. Operations for execution on the GPU are coded in terms of a “GPU kernel.” Kernel execution is handled by thread blocks, where each thread block is usually composed of multiple threads. In the current compute capability, a thread block can contain up to 1024 individual threads, arranged in up to three dimensions [12]. Usually, one thread block is not sufficient to cover the complete data stream. Instead, multiple thread blocks are arranged in an up to three-dimensional grid. For good performance, it is essential to find a trade-off between the thread-block size and the number of thread blocks forming a grid: few large thread-blocks allow for better data reuse through shared memory, smaller thread blocks give the scheduler more flexibility in assigning thread blocks to the multiprocessors. In case of using small thread-block sizes, a multiprocessor may schedule multiple thread-blocks in parallel.

C. *GPU execution mode*

Every SMX of the GK110 architecture can manage up to 2048 threads, arranged in from anywhere from 2 to 16 different thread-blocks. The upper bound of 16 is a strict hardware limit, the lower bound of 2 is due to a maximum of 1024 threads forming one thread block. The number of threads that is executed in parallel is, however, significantly smaller, and depends on the hardware characteristics and the kernel requirements such as registers, shared memory, or thread-block size. The intention of having a larger number of threads active is to quickly switch in-between them to cover memory latency. E.g., if threads in a warp issue a memory operation, those threads will stall while waiting for the memory transaction to complete. To combat this, rather than allowing the hardware to stall, the warp scheduler may find a warp that is not waiting for a memory operation to complete, and begin executing this warp instead. This constant juggling of active warps allows the GPU to tolerate the high memory latency and attempt to keep the compute cores occupied. A result of this operation mode, the data will not be handled in parallel for data streams that are too large for one warp; instead some data has to wait until resources are available again. This issue leads us to consider the execution mode as block-asynchronous, consistent with the notation for iteration schemes, where local, synchronous iterations are embedded in a global asynchronous iteration [2]. Using this interpretation, a central question is what size the blocks of synchronous, parallel execution have. While the technical characteristics of the GPU hardware may suggest that the blocks of parallel execution are consistent with the warps, we will investigate this question experimentally the Section IV.

```
__global__ void kernel1( int n, double *val ) {
    int i = blockDim.x + blockIdx.x + threadIdx.x;
    if ( (i > 0) && (i < n-1) )
        if( val[ i+1 ] == 1.0 && val[ i-1 ] == 1.0 )
            val[ i ] = 0.0;
}
```

Figure 3: CUDA kernel revealing block-asynchronous execution scheme of GPUs.

```
__global__ void kernel2( int n, double *val ) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int tid = threadIdx.x;
    __shared__ double tmp[ BLOCKSIZE ];
    if ( i<n )
       tmp[ tid ] = val[ i ];
    __syncthreads();
    if ( (tid > 0) && (tid < BLOCKSIZE-1) ){
        if( tmp[ i+1 ] == 1.0 && tmp[ i-1 ] == 1.0 )
            tmp[ i ] = 0.0;
    } else if( (tid == 0) && (i > 0) ){
        if( val[ i-1 ] == 1.0 )
            tmp[ i ] = 0.0;
    } else if( (tid == BLOCKSIZE-1) && (i < n-1) ){
        if( val[ i-1 ] == 1.0 )
            tmp[ i ] = 0.0;
    }
    __syncthreads();
    if ( i<n )
       val[ i ] = tmp[ tid ];
}
```

Figure 4: CUDA kernel revealing block-asynchronous execution scheme of GPUs.

IV. Experimental Analysis

In this section we experimentally analyze the effects of the GPU execution scheme. The analysis is based on kernels operating on a one-dimensional data stream that we call a "vector" consisting of "components." Although the findings also hold for multi-dimensional data streams and higher thread block dimensions, we consider only a setting where the compute thread are arranged linearly within each thread block, and the thread blocks are arranged linearly within the compute grid. This simplification corresponds to a one-dimensional grid containing one-dimensional thread blocks. Also, it allows for mapping threads to vector components.

A. *Evaluation paradigms*

With the goal of relating the block-asynchronous execution mode to the mathematical perception of block-asynchronous updates, we first have to define a suitable concept of parallel execution. Given a data set, we call the update of two components a "parallel update" if the following conditions are fulfilled:

1) The update process of the two components is based on an identical memory state for all other components.
2) The update process of the two components is based on the pre-update memory state of the respectively other component.

For dependent update operations, this concept of parallelism implies that two updates are not considered parallel if one component is updated based on an already updated memory state of the other component. We note that we are in particular interested in determining whether components adjacent in main memory are updated in parallel or non-parallel fashion. This allows to identify sets of consecutive components that are handled in parallel.

In general, it is very difficult to identify the largest coalesced entity for which parallel updates can be expected. Component updates are handled by threads, the threads are aggregated in thread blocks, and distinct thread blocks may be assigned to the same, or different multiprocessors. Also, the thread block scheduling an the scheduling of warps within a thread block is not deterministic, and can, theoretically, differ between runs. The approach we use to identify regions of parallel execution is based on a concept we call "memory inconsistencies". We define a memory inconsistency the situation where a component update is executed at a point when a component adjacent in memory has already been updated. This concept accounts for all memory levels, i.e. we also call it a memory inconsistency if the adjacent component is not yet updated in main memory, but in local multiprocessor memory. Hence, memory inconsistencies are determined by what values an update operation would read for its neighbors.

For experimental investigation, we initiate a vector with ones, and design a simple kernel that sets vector values to zero if both adjacent values are one, see **kernel1** in Figure 3. If the kernel reads an already updated value for one of its immediate neighbors, the vector component remains one. During kernel execution, the vector values are read, updated, and written back to the GPU main memory. The output vector is expected to show a pattern: zeros corresponding to regions of parallel updates, ones indicating locations where adjacent vector components where handled subsequently.

The same detection strategy is used in a second kernel, that, oppose to **kernel1**, explicitly uses the local multiprocessor memory as intermediate storage for reading, updating, and writing the values (see **kernel2** in Figure 4). Synchronizations separate these data access operations from

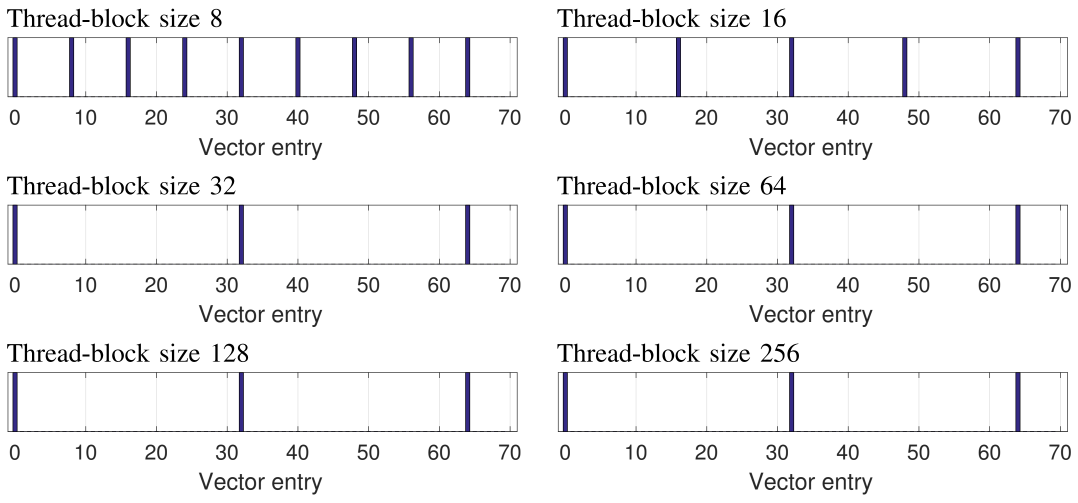

Figure 5: Detected memory inconsistencies in the vector updates. Values larger zero indicate locations where a vector value was updated subsequently to one of its intermediate neighbors. Zero values correspond to regions where no memory inconsistencies occurred.

the actual component update. The intention of disconnecting the data read and write phases from the computation is to determine the parallelism level used for the component updates.

As previously mentioned, the GPU kernel will be executed in terms of thread blocks. The order in which the GPU thread blocks are executed is not deterministic, and can, theoretically, differ between runs. The same holds for the execution of the distinct warps within on thread block. Therefore, we base the analysis on a high number of kernel executions. The goal is to catch all memory inconsistencies that can potentially occur. Although the experiments are based on 1,000,000 kernel executions, we emphasize that the experimental analysis only allows to identify memory inconsistencies, it is impossible to identify areas where parallel execution is guaranteed. Hardware characteristics and experimental results may suggest those regions, but no experiment allows for a conclusive proof of this.

B. *Experimental results*

In a first experiment, we run **kernel1** with different thread block sizes. We then map all detected memory inconsistencies to the first set of thread blocks. Precisely, if a memory inconsistency was encountered at a location i, we map this inconsistency to location j where $j = mod_{256}(i)$, and 256 being a multiple of all evaluated thread block sizes. This helps in hiding effects associated with the scheduling of the first thread blocks.

Figure 5 visualizes the memory inconsistencies we encounter when executing **kernel1**. The first row of Figure 5 visualizes the inconsistencies for thread-block sizes 8 and 16, which is smaller than the warp size (32). Memory inconsistencies occur at the thread-block boundaries. Within a thread block, no memory inconsistencies are detected. The second row shows the data for thread-block sizes 32 and 64. 32 is also the warp size. The inconsistencies at the thread-block boundaries are expected. For thread-block size 64 (right figure in second row), memory inconsistencies occur not only at the thread-block boundaries (component 0 and 64), but also at the location 32. This indicates that components handled by different warps were, although part of the same thread block, not updated in parallel fashion. The same pattern can be observed for thread-block sizes 128 and 256, see the plots in the third row of Figure 5.

The experimental results are consistent with the GPU characteristics. Thread-block sizes larger the warp size ensure the vector components are handled by the same multiprocessor, however not necessarily in parallel. **kernel1** composes of memory reads, component updates, and memory writes, and with the memory access being handled at the granularity of warp, a multiprocessor may interleave read, update and write phases of distinct warps to hide memory latency.

Another interesting observation from Figure 5 is that the detected memory inconsistencies are all located in the first component of a thread block or warp. For this kernel, we never encounter the situation where the last component in a warp or thread block is updated after a immediate neighbor has been updated. This indicates that in this experiment, the thread blocks were – in particular if handled by the same multiprocessor – executed in increasing order. If handled by different thread blocks, the exact update order may have been hidden by the delay of accessing main memory.

Next, we investigate the impact of using shared memory

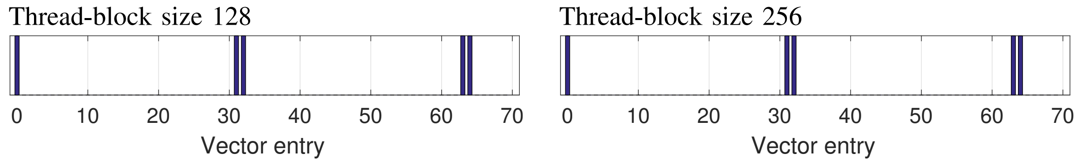

Figure 6: Update inconsistencies in the vector updates using shared memory with a synchronization barriers inside each thread block before and after data transfers.

for thread block configurations exceeding the warp size. The first explicit synchronizations in **kernel2** enforce that all data is loaded into multiprocessor memory before any component (of the thread block) is updated (see line 13 Figure 4). The second synchronization (line 30 in Figure 4) ensures that all components are updated in shared memory before any component is written back to main memory. This isolates the component update from the main memory access. The results obtained from executing **kernel2** with thread block size configurations 128 and 256 are visualized in Figure 6. Again, we detect memory inconsistencies not only at the thread block boundaries, but also at the warp boundaries. This shows that also the arithmetic operations in the component update phase uses a parallelism level of the warp size.

A difference to the results in Figure 5 is that update inconsistencies are not only detected at the beginning of the warps (i.e. components 0, 32, 64), but also in the last components of a warp (components 31, 63). This is only possible if a component with a lower ID is updated at a point where an adjacent component with a higher ID has already been updated, i.e. warp with higher ID has completed the component update – at least in the local multiprocessor memory – before a warp with a lower ID. We conclude that the previously observed linear consecutive scheduling of warps may be violated.

In a nutshell, the key observations from running **kernel1** and **kernel2** are:

- If the data dependencies allow, thread blocks are preferably scheduled in consecutive, increasing order.
- For thread-block sizes smaller or equal the warp size, we have no indications of non-parallel execution within a thread block.
- For thread block sizes exceeding the warp size, components assigned to distinct warps may be handled subsequently.
- With the intention of hiding memory latency, the read, update, and write phases of distinct warps may be interleaved. For thread block sizes larger the warp size, this can result in a situation where components part of the same thread block are handled based on different stages of the local multiprocessor memory.
- Memory operations and arithmetic operations are executed at a parallelism level of the warp size, which is 32 for the current NVIDIA GPU hardware.

C. Summary

This study investigated the block-asynchronous execution mode of NVIDIA's GK110 GPU processor.

We have seen that components assigned to distinct warps may be handled subsequently. In particular, increasing the thread block size beyond the warp size does not result in larger sets of consecutive components that are handled in parallel. For thread block sizes smaller the warp size, the thread-block to warp mapping is non-deterministic, and components assigned to different thread blocks may be handled subsequently. These observations show that the subset size of the synchronous inner iterations like they occur in the block-asynchronous fixed-point iterations considered in [2], [3], [4], [9] is limited by the warp size 32.

Acknowledgment

The authors would like to acknowledge support from the U.S. Department of Energy (Award Number DE-SC-0010042), and NVIDIA. Furthermore, the authors would like to thank Jeff Larkin for many fruitful technical discussions, and for providing helpful comments on an earlier version of the manuscript.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[2] H. Anzt, "Asynchronous and Multiprecision Linear Solvers - Scalable and Fault-Tolerant Numerics for Energy Efficient High Performance Computing," Ph.D. dissertation, Karlsruhe Institute of Technology, Institute for Applied and Numerical Mathematics, Nov. 2012.

[3] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, "A block-asynchronous relaxation method for graphics processing units," *Journal on Parallel Distributed Computing*, vol. 73, no. 12, pp. 1613–1626, 2013.

[4] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline, "Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems," in *ICCS*, ser. Procedia Computer Science, vol. 9. Elsevier, 2012, pp. 7–16.

[5] H. Anzt, P. Luszczek, J. Dongarra, and V. Heuveline, “GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement,” in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 908–919.

[6] M. Wlotzka and V. Heuveline, “Block-asynchronous and jacobi smoothers for a multigrid solver on gpu-accelerated hpc clusters,” University of Heidelberg, Preprint Series of the Engineering Mathematics and Computing Lab (EMCL) 03, 2015.

[7] M. R. Rodriguez, B. Philip, Z. Wang, and M. A. Berrill, “Block-relaxation methods for 3d constant-coefficient stencils on gpus and multicore cpus,” *CoRR*, vol. abs/1208.1975, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1208.html#abs-1208-1975

[8] H. Anzt, E. Chow, and J. Dongarra, “Iterative sparse triangular solves for preconditioning,” in *Euro-Par 2015: Parallel Processing*, ser. Lecture Notes in Computer Science, 2015, vol. 9233, pp. 650–661.

[9] E. Chow, H. Anzt, and J. Dongarra, “Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs,” in *Lecture Notes in Computer Science*, vol. 9137, July 12 – 16 2015, pp. 1–16.

[10] NVIDIA Corporation, “Kepler GK110 whitepaper,” 2012.

[11] *NVIDIA CUDA Toolkit*, 7th ed., NVIDIA Corporation, March 2015.

[12] *CUDA Toolkit v7.5*, NVIDIA Corporation, September 2015.