2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00068

CUDAMicroBench: Microbenchmarks to Assist CUDA Performance Programming

Xinyao Yi
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
xyi2@uncc.edu

David Stokes
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
dstokes5@uncc.edu

Yonghong Yan
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
yyan7@uncc.edu

Chunhua Liao
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550, USA
liao6@llnl.gov

***Abstract*—Programming to achieve high performance for NVIDIA GPUs using CUDA has been known to be challenging. A GPU has hundreds or thousands of cores that a program must exhibit sufficient parallelism to achieve maximum GPU utilization. A system with GPU accelerators has a heterogeneous and deep memory system that programmers must effectively and correctly use to fully take advantage of the GPU's parallelism capability. In this paper, we present CUDAMicroBench, a collection of fourteen microbenchmarks that demonstrate performance challenges in CUDA programming and techniques to optimize the CUDA programs to address these challenges. It also includes examples and techniques for using advanced CUDA features such as data shuffling between threads, dynamic parallelism, etc that can help users optimize the CUDA program for performance. The microbenchmark can be used for evaluating the performance of GPU architectures, the memory systems of GPU itself and of the whole system architectures, and for evaluating the effectiveness of compiler and performance tools for performance analysis. It can be used to help users understand the complexity of heterogeneous GPU-accelerator systems through examples and guide users for performance optimization. It is released as BSD-licensed open-source from https://github.com/passlab/CUDAMicroBench.git.**

***Index Terms*—GPU, CUDA, Performance Optimization, Parallelism, Memory Hierarchy**

I. Introduction

The GPU manycore architecture excels at data-parallel computation for performance and energy efficiency because of the massive parallel-processing capabilities of GPUs. A highly-optimized GPU program can achieve 10x to 100x performance improvement over its CPU versions. A computer system with GPU accelerators, e.g. those from NVIDIA, incorporates the GPU devices to augment the performance of conventional multicore processors. Such a system has a heterogeneous and deep memory system that programmers can explicitly manage the data movement between host memory and device memory. For programming, offloading is the model used by most GPU programming languages, i.e. computational loops, known as kernels, and the input data of the kernel need to be offloaded onto an accelerator in order to perform computation.

The performance advantage of GPU accelerated systems, and the complexity of its parallel architecture and heterogeneous and deep memory system come together as a double-edged sword for average users since programming to achieve high performance has been known to be challenging. In this paper, we summarize three important guidelines for developing high-performance GPU programs: 1) optimizing kernels to saturate the massive parallel capability of GPUs, 2) effectively leveraging the deep memory hierarchy inside GPU to maximize its computing efficiency for kernel execution, and 3) properly arranging data movement between CPU and GPU to reduce the performance impact of data movement for the GPU program. Under these guidelines, we collect a set of microbenchmarks that exhibit the performance challenges in CUDA programming. These microbenchmarks also include techniques to optimize a CUDA program to address the challenges.

Other than the developed microbenchmarks, the contributions of this paper also include: 1) provide performance optimization techniques for using memories on GPU and between CPU and GPU, such as coalesced and aligned memory access, use of read-only memory, impact of access density to the performance; and 2) provide performance analysis to demonstrate the impact of those problems to application performance using the developed microbenchmarks. The microbenchmarks can be used to evaluate the efficiency of different parallel programming techniques, evaluate the performance of memory systems of different NVIDIA GPUs and memory system architectures, and evaluate software tools for performance analysis. It can be used to help users understand the complexity of heterogeneous GPU-accelerator systems and guide users for performance optimization.

In the rest of the paper, we describe the manycore architectures and the heterogeneous memory system of the GPU accelerated system in Section II. Then the benchmarks are presented in three classes according to the three guidelines in

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00068

Section III, IV and Section V. In Section VI, related work are discussed. The paper is concluded in Section VII.

II. Motivation and Benchmark Summary

In this section, we describe the GPU manycore architecture, the memory systems of GPU itself, and of a GPU-accelerator system to indicate the challenges of performance optimization of programming GPU.

A. The NVIDIA GPU Manycore Architecture

The massive parallelism capability of GPU is realized with its manycore architecture. The newest Nvidia's Ampere A100 GPU contains 108 streaming multiprocessors(SM), each containing four Texture Mapping Units that each contains 16 INT32 ALU's, 16 FP32 ALUs, and 8 FP64 ALUs, and one tensor core. Thus in total, the number of cores of GPU is above 5000. GPU manycore architecture implements a thread execution model known as the single instruction multiple thread (SIMT). With SIMT, the GPU executes instructions in lock-step across multiple threads that process different data. GPU schedules warps of threads, typically 32 threads in each warp, onto its many cores, and warp is the scheduling unit of kernel execution. For programming, the NVIDIA CUDA threading model allows for a much large number of threads, e.g. millions of SIMT threads and threads are organized in a two-level (block and grid) hierarchy. In each level, i.e threads of a block or blocks of the grid, can be structured in 1-D, 2-D or 3-D topology. Fig. 1 shows an example of the CUDA threading model. Programmers can determine how to divide computation work and data under this SIMT threading model to fully utilize GPU cores.

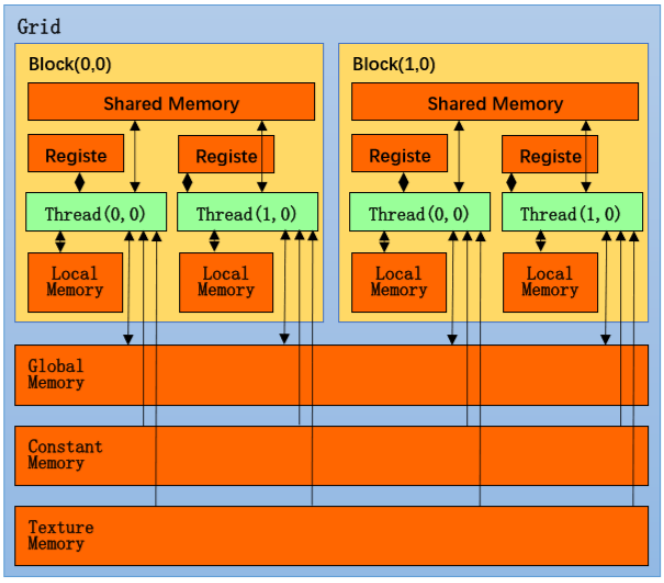


Fig. 1: NVIDIA CUDA threading model and GPU internal memory

B. Two Memory Systems of a GPU-Accelerated Computer

A GPU-accelerated computer system can be considered to have two memory systems, the internal memory hierarchy of the GPU itself, and the discrete memory system that both CPUs, GPUs and other devices on the system leverage. Fig. 1 shows the internal memory hierarchy of a typical NVIDIA GPU, which is more complex than CPU memory hierarchy. Registers,local memory and shared memory are on-ship memories while local, global, constant, and texture memory all reside off-chip. There are significant trade-offs depending on the type of memory used. Off-chip memories can be addressed by any block or threads, while on-chip memories are limited in scope to the specific block that they are a part of.

The whole memory system of a GPU-accelerated computer is a discrete memory system that requires explicit communication between the GPU and CPU for data movement. Memory allocation and data movement are initiated from the CPU explicitly. The primary challenge with this is the difficulty of efficiently leveraging explicit control of the memory system.

The difficulty involved with programming the discrete memory system led to the introduction of unified memory in CUDA 6.0, which sought to simplify memory management for programmers. The unified memory system allows for a single allocation call to be made with cudaMallocManaged for memory segment that is accessible by both CPU and GPU. While memory operations are transparent to the programmer, the hardware is still fundamentally discrete and data copy operations happens via paging mechanisms by the driver. The automation of these memory operations can be inefficient, leading to reduced performance as shown in our evaluation.

The manycore architecture and the complexity of GPU heterogeneous memory architecture and system indicate three important guidelines for developing high-performance CUDA programs. First, GPU kernels should be optimized to saturate as much as possible the massive parallel capability of GPUs. Second, deep memory hierarchy inside GPU should be effectively leveraged to maximize the computing efficiency of GPU for kernel execution. Third, memory management and data movement between CPU and GPU memory should be properly arranged to reduce the performance impact of data movement operations.

To assist programmers for CUDA performance programming according to the three mentioned guidelines, we believe small and representative examples to demonstrate the challenges and techniques are an effective approach to help programmers. We present in this paper 14 specific techniques and illustrate them with small code examples that collectively we refer to as CUDAMicroBench. These microbenchmarks are categorized into three classes according to the three guidelines, summarized in Table I. All tests are performed on Carina (Intel® Xeon® Gold 6230N 2.30GHz CPU and NVIDIA Tesla V100) and Fornax (Intel® Xeon® E5-2699 v3 2.3GHz CPU and NVIDIA K80) except asynchronous copy of global to shared memory in IV-D, which used an NVIDIA RTX 3080 GPU.

TABLE I: Summary of the CUDAMicroBench microbenchmarks, https://github.com/passlab/CUDAMicroBench.git

Benchmark	Pattern of Performance Inefficiency	Optimization techniques	Speedup	Programmability
Optimizing Kernels to Saturate the Massive Parallel Capability of GPUs				
WarpDivRedux	Threads enter different branches when they encounter the control flow statement	Change the algorithm: take the warp size as the step	1.1 (average)	3
DynParallel	Workloads that require the use of nested parallelism such as those using adaptive grids	Use dynamic parallelism to allow the GPU to generate its own work	3.26 (best)	4
Conkernels	Launch multiple kernel instances on one GPU	Use concurrent kernels technique	7 (average)	4
TaskGraph	Provide a more effective model for submitting work to the GPU	Pre-define the task graph and run-repeatedly execution flow	Only for easier programming	3
Effectively Leveraging the Deep Memory Hierarchy Inside GPU to Maximize The Computing Capability of GPU for Kernel Execution				
Shmem	The data need to be accessed several times	Use shared memory to store the data which needs to be accessed repeatedly	1.25 (average)	2
CoMem	Stride or random access of array across threads which have uncoalesced memory access	Consecutive memory access across threads	18 (average)	3
MemAlign	Memory are allocated at unaligned address	Use aligned malloc	1.1 (average)	1
GSOverlap	Global-shared memory copy takes much time	Use the new function *memcpy_async* in CUDA11 to accelerate the data transfer	1.04 (best)	3
Shuffle	The data exchange between threads	Use shuffle to enable threads in the same warp directly share part of their results between registers	1.25 (average)	5
BankRedux	Two or more threads access different locations of the same bank	Change the algorithm to avoid bank conflicts	1.3 (average)	5
Properly Arranging Data Movement Between CPU and GPU to Reduce the Performance Impact of Data Movement for the GPU Program				
HDOverlap	Host-device memory copy takes much time	Use *cudaMemcpyAsync* function to accelerate the data transfer	1.036(best)	1
ReadOnlyMem	Large amount of read-only data	Put read-only data in constant/texture memory to get a higher speed	4.3 (best)	1
UniMem	Low memory access density	Put the data in unified memory and only copy the necessary pages	3 (average)	3
MiniTransfer	Wrong data layout causes a large amount of useless data transfer between CPU and GPU	Change to the correct data layout to avoid useless data transfer	190 (best)	5

[a] The "programmability" column is used to indicate the difficulty of programming for a given optimization in our opinion, on a scale from 1 to 5 with 5 being the most difficult.

III. Optimizing Kernels to Saturate the Massive Parallel Capability of GPUs

For the first objective of optimizing GPU kernels to saturate the parallel capability of GPUs, we include four techniques for CUDA performance optimization. For each technique, we include a short description, discussion of its benefits, potential use cases, and a microbenchmark to demonstrate.

A. Warp Divergence

A warp is the scheduling unit of GPU kernel execution, made up of a group of threads within a block. Because of the lock-step SIMT execution model of GPU, when two threads in a warp execute two separate branches during execution, there could be significant performance penalty. This divergence of threads because of the use of branching is known as warp divergence. Fig. 2 shows two examples. For the first kernel, different operations are performed according to the parity of the thread ID. It is intended that the even- and odd-numbered threads in a warp execute the true and false branches respectively of the if statement. However, because of the SIMT lock-step execution model, all warp threads execute both branches, though only the relevant threads commit the results. This execution model for branches result in performance loss.

The second kernel shows how to optimize the first one to remove warp divergence. It ensures that all threads in a warp execute the same branch. The performance results are shown in Fig. 3. The execution efficiency of the two kernels, produced using nvprof, are 85.71% vs 100%, which correspond to the performance results in the figure.

```
1  __global__ void WD(float *x,
2    float *y, float *z) {
3    int tid=blockIdx.x*blockDim.x+threadIdx.x;
4    if (tid%2 == 0) {
5      z[tid] = 2 * x[tid] + 3 * y[tid];
6    } else {
7      z[tid] = 3 * x[tid] + 2 * y[tid];
8    }
9  }
10
11 __global__ void noWD(float *x,
12   float *y, float *z) {
13   int tid=blockIdx.x*blockDim.x+threadIdx.x;
14   if ((tid / warpSize) % 2 == 0) {
15     z[tid] = 2 * x[tid] + 3 * y[tid];
16   } else {
17     z[tid] = 3 * x[tid] + 2 * y[tid];
18   }
19 }
```

Fig. 2: Warp divergence example kernels

B. Dynamic Parallelism

Dynamic parallelism was introduced in CUDA 5.0, supported by devices with compute capability 3.5 and higher. The feature allows the GPU to generate its own work by launching another kernels instead of requiring the kernel jobs to be submitted by the CPU. It allows for any given thread in a block to launch its own child block. Workloads that require the use of nested parallelism such as those using adaptive grids particularly benefit from the feature. For example, it facilitates the implementation of adaptive refinement for fine grained detail to be computed for certain areas that require high

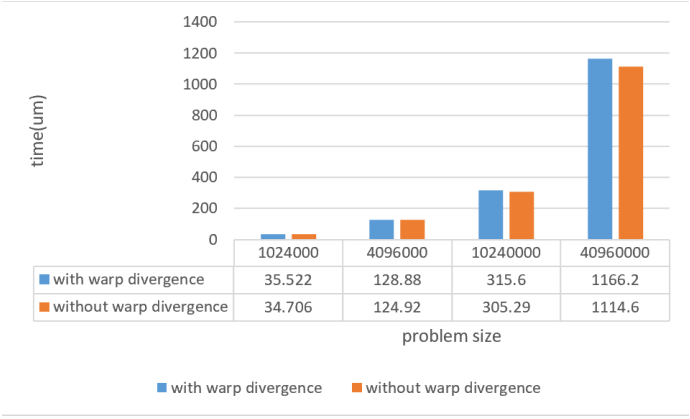


Fig. 3: Performance of warp divergence tests on NVIDIA Tesla V100 GPUs

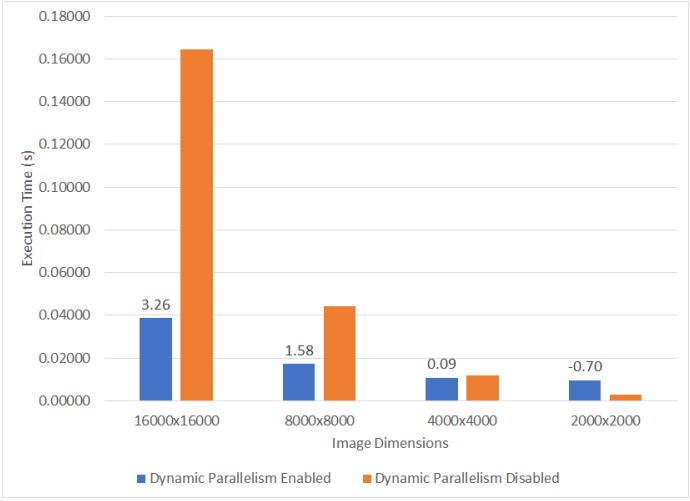


Fig. 5: Performance of dynamic parallelism on NVIDIA RTX 3080 GPUs

fidelity. Other workloads that benefit are recursive algorithms and those that need large independent batch processes.

```
__global__ void mandelbrot_block_k() {
  int comm_dwell = border_dwell();
  if (Perimeter dwells equal) {
    dwell_fill_k <<<grid, bs >>>();
  } else if (not hit subdivision limit
             and not hit depth limit) {
    //subdivide recursively
    mandelbrot_block_k <<<grid, bs >>>();
  } else {//do fine grain per pixel
    pixel_calc<<<grid, bs>>>();
  }
}
```

Fig. 4: Nested kernel using dynamic parallelism

We use the Mandelbrot set example [1] shown in Fig. 4 as the microbenchmark to demonstrate the feature of dynamic parallelism. The pseudo code is a portion of the Mariani-Silver algorithm, which recursively subdivides the calculations for the Mandelbrot set. The performance of this algorithm was compared against the escape time algorithm which is considered the standard solution. The basic operation is the same for calculating a pixel, so the results are comparable as Mariani-Silver is able to avoid a portion of computation. The results show significant improvement of 3.26x times when using dynamic parallelism over disabling it for rendering a 16000x16000 image as shown in Fig. 5. The speedup is shown above the bar for enabling dynamic parallelism. It is also shown that the performance improvement decreases as the size of the image decreases. This is demonstrated by the results of a 2000x2000 image, where overhead of dynamic parallelism outweighs the benefit of using it.

C. Concurrent Kernels

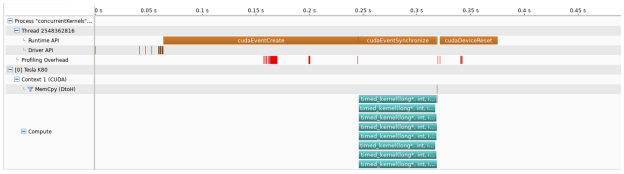
From Fermi architecture, NVIDIA GPUs introduced a feature called "concurrent kernels", which enable CPU to launch multiple kernel instances on one GPU. This feature helps performance improvement for those kernels that are memory bounds, because such algorithms usually do not exhibit sufficient parallelism, and allowing executing multiple kernels concurrently would increase utilization of GPU cores.

To demonstrate this feature, we select the example from CUDA Samples [2] as our microbenchmarks. In this program, there are multiple asynchronous kernels that are to be launched by a loop and each kernel is associated with a CUDA stream. The concurrent execution of those kernels can be visualized using the NVIDIA nvvp thread management tool, which is used to observe threads' activities visually of CUDA kernels. Fig. 6 shows the results of this example. When we use 8 CPU threads to launch concurrent kernels, the version that uses concurrent kernel execution is approximately 7 times faster than the version that uses serial kernel launching. This indicates the effect of concurrent kernel execution. It can be seen that the high degree of thread concurrency has a significant impact on performance, and increasing the thread concurrency as much as possible can greatly improve the execution efficiency of the program.

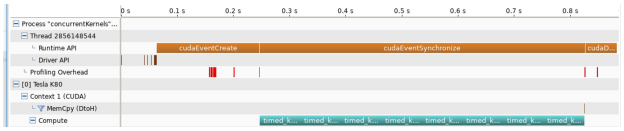
D. Task Graph

The task graph feature was introduced in the 2018 release of CUDA 10. We include a code example from CUDA-Samples [2] to help illustrate the use of this feature. Since this feature is intended mainly for help improving programmability, we do not provide performance study.

This feature allows for a more effective and flexible model for submitting work to the GPU. The task graph can consist of a series of operations, such as memory copy and kernel startup, which are connected through dependencies and defined separately from their execution. It provides a mechanism to launch multiple GPU operations through a single CPU operation to reduce overheads. In some specific applications which has many repeatable tasks, task graph might improve the application's efficiency and performance.



(a)



(b)

Fig. 6: The activities of kernels with and without using concurrent kernels feature on NVIDIA Tesla V100 GPUs: (a) using concurrent kernels, and (b) not using concurrent kernels.

IV. Effectively Leveraging the Deep Memory Hierarchy inside GPU

In this subsection, we describe microbenchmarks for performance optimization according to the second guideline, which is to effectively leverage the deep memory hierarchy inside GPU. We demonstrate the challenges of using GPU memory using microbenchmarks and the optimized version of the microbenchmarks are included that addresses the challenges.

A. Using Shared Memory to Improve the Performance

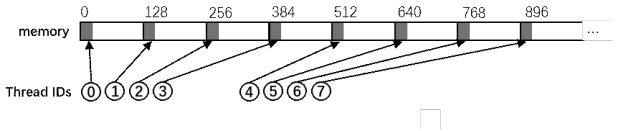
Shared memory is a high-speed programmable SRAM on the GPU chip, and all threads in the same block can access it. The latency of the shared memory is almost the same as that of the register, and its capacity is several times that of the register. Shared memory is mostly used as programmable cache for frequently accessed data items of the GPU kernels.

A known example of using shared memory for GPU performance optimization is matrix multiplication which has a very high data reuse rate. The example is also available from CUDA Samples or the CUDA developer manual and we include it as the microbenchmark. In the implementation, the algorithm divides the large matrix into 16x16 tiles, copies each tiles into the shared memory for calculation, and then performs the accumulation operation. For a matrix with a size of 2048*2048, compared to the version that only uses global memory, using shared memory can increase performance by 20% and scales well with matrix size on most NVIDIA GPUs.

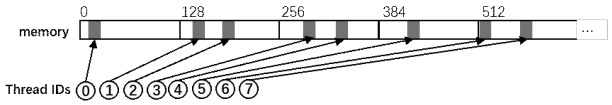
B. GPU Coalesced and Uncoalesced Memory Access

On a GPU, data transfer between global memory and on-chip storage are by chunk for each memory transaction, even only a small subset of a chunk is requested by a thread. Memory coalescing is a technique in GPU programming to use minimum transactions to fulfill the memory requests by a large number of threads. This is accomplished by combining references to adjacent memory locations of multiple threads into minimum number of transactions. Fig. 7 provides a pictorial view of coalescing and uncoalescing of memory access by 8 threads accessing 128-byte of data that is the size of a chunk of a memory transaction. Fig. 7 (a) shows that 8 threads access the consecutive bytes of the 128 bytes, thus the memory requests by all 8 threads can be fulfilled by one memory transaction. Fig. 7 (b) shows the situation in which the 8 threads access the bytes with 128-byte stride. 8 memory transactions are needed to transfer 8 * 128 bytes to fulfill the data requests (only 128 bytes) by the threads. The third figure shows the random access situation that causes uncoalesced memory access. Eight adjacent threads need to access memory with unevenly distributed strides. Five memory transactions are needed to transfer the data needed by the threads.

(a)



(b)



(c)

Fig. 7: GPU coalesced and uncoalesced memory access: a) coalesced memory access by 8 threads, b) strided memory access and uncoalesced, c) random memory access and uncoalesced

When writing CUDA programs, the approach of how the iteration of a data-parallel loop is distributed to the threads impacts the memory access pattern of CUDA threads, thus could result in coalesced or uncoalesced memory access. There are two approaches that are commonly used for loop distribution, block and cyclic distributions. We use the AXPY kernel shown in Fig. 8 to demonstrate this. AXPY calculates Y[i] += a * X[i] for N number of array elements of X and Y.

As shown in the second kernel, a block distribution splits the iterations into chunks of contiguous iterations among all threads, one thread per chunk. It will end up as uncoalesced memory access by the threads to the array X and Y. The third kernel shows the cyclic distribution which can fix the uncoalesced memory access. Loop iterations are assigned to threads in cycles starting from the first iteration for one iteration per thread. Threads are recycled for the remaining iterations. By doing this, we allow threads to access consecutive elements of the two arrays and coalesced access is achieved.

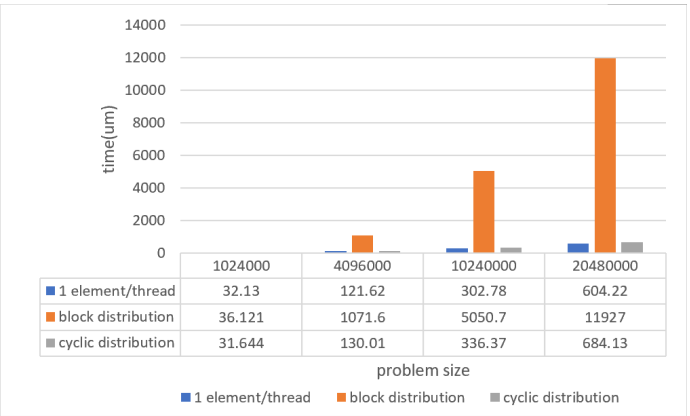
The performance impact of the two distributions is shown in Fig. 9 which shows that using cyclic distribution is about 18 times faster than using block distribution.

```
__global__ void axpy_1perThread(REAL* x,
  REAL* y, int n, REAL a) {
  int i=blockDim.x*blockIdx.x+threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

/* block distribution of loop iteration */
__global__ void axpy_block(REAL* x,
  REAL* y, int n, REAL a) {
  int i=threadIdx.x+blockIdx.x*blockDim.x;
  int total_threads=gridDim.x*blockDim.x;
  int block_size=n/total_threads;
  int start_index=i*block_size;
  int stop_index=start_index+block_size;
  for (j=start_index; j<stop_index; j++)
    if (j < n) y[j] += a*x[j];
}

/* cyclic distribution of loop iteration */
__global__ void axpy_cyclic(REAL* x,
  REAL* y, int n, REAL a) {
  int i=threadIdx.x+blockIdx.x*blockDim.x;
  int total_threads=gridDim.x*blockDim.x;
  for (j=i; j<n; j+=total_threads)
    if (j < n) y[j] += a*x[j];
}
```

Fig. 8: AXPY kernels that exhibit coalesced and uncoalesced memory access



	1024000	4096000	10240000	20480000
1 element/thread	32.13	121.62	302.78	604.22
block distribution	36.121	1071.6	5050.7	11927
cyclic distribution	31.644	130.01	336.37	684.13

Fig. 9: AXPY kernel execution time that shows this performance difference between coalesced and uncoalesced memory access on NVIDIA Tesla V100 GPUs. The kernel configuration for BLOCK and CYCLIC is $<<< 1024, 256 >>>$.

For the example that has uncoalesced memory access caused by random memory access of consecutive threads shown in Fig. 7 (c), we include sparse matrix multiplication in CUDAMicroBench. If not using proper condensed format of the multiplier matrix, e.g. compressed sparse row (CSR) format or compressed sparse column (CSC) format, uncoalesced memory access would occur, degrading the performance. The correct way to optimize this is to use the right combination of CSR and CSC for the multiplier and final matrices.

C. Memory Alignment for the Memory Access of GPU Kernels

As mentioned before, the GPU memory controller transfers memory in chunks for each memory transaction, e.g. 128-byte chunk per transaction. Aligned memory access means that the first memory address accessed is the exact multiple of a memory chunk. Aligned and misaligned memory access can be demonstrated by modifying the AXPY example, shown in Fig. 10. The results show that the aligned access has a clear though small performance improvement (about 3%), because of smaller number of memory transactions are performed in the aligned memory access situation than in the misaligned memory access situation, for the same amount of needed data.

```
__global__ void axpy_aligned(REAL* x,
  REAL* y, int n, REAL a) {
  int i=blockDim.x*blockIdx.x+threadIdx.x;
  if (i > 0 &&i < n) y[i] += a*x[i];
}
__global__ void axpy_misaligned(REAL* x,
  REAL* y, int n, REAL a) {
  int i=blockDim.x*blockIdx.x+threadIdx.x+1;
  if (i < n) y[i] += a*x[i];
}
```

Fig. 10: Kernels of AXPY that have aligned and misaligned memory accesses. (a) aligned memory access in which a warp of 32 threads request 256 bytes of data fulfilled by two 128-byte memory transactions; (b) misaligned memory access. The 256 bytes of data needed by a warp of threads need three 128-byte memory transactions because of the misalignment.

The reason for the minor performance difference of the two memory accesses is that the GPU (Tesla V100) has L1 cache, making the effect of misalignment on throughput for sequential memory accesses across threads negligible. However, for some GPUs without L1 cache, the performance loss caused by misaligned access could be much larger. For example, for NVIDIA GPU with computing power of 1.0 which only has an L2 cache, any unaligned accesses twisted by the half-thread will result in 16 separate 32-byte transactions.

D. Overlapping and Pipelining Data Copy Between Global Memory and Shared Memory Using Memcpy_async

As we mentioned at the beginning of this section, shared memory has often been used as programmable software cache for keeping frequently-used items in the fast memory. Before using it, data needs to be copied from slower global memory to the faster shared memory. To accelerate this copy, recent CUDA introduces asynchronous memcpy, the *memcpy_async* function, for further optimize data movement between global memory and shared memory. This feature is realized based on two aspects in NVIDIA's new Ampere Architecture, being the hardware acceleration of the *memcpy_async* operation that bypassing register access, and pipelining the *memcpy_async* operation to overlap computation and memory operations.

The hardware acceleration of the memory operation is done by bypassing temporary registers when copying from

global to shared memory. This hardware level optimization provides a small but meaningful performance improvement. This performance improvement is demonstrated in AXPY with an approximately 5 millisecond difference in execution time on NVIDIA Tesla RTX 3080 GPU, meaning the asynchronous kernel is 1.04 times faster. This can be expected to increase as the amount of reads and writes to the shared memory increases allowing for further scaling. This feature is still new and as such will likely have further research conducted that will provide greater insight into the optimal implementation.

E. Data Shuffle Between Threads

Before the Kepler architecture of NVIDIA GPUs, the data exchange between threads must go through shared memory. For programs that frequently exchange data between threads, the shared memory bandwidth will undoubtedly become a performance bottleneck. Kepler and new generation GPUs introduce a new warp-level intrinsic called the shuffle operation. This feature allows threads in the same warp to exchange data directly between registers, bypassing the local memory.

We include a reduction microbenchmark to show the advantages of using shuffle [3]. Compared with the traditional version, by using shuffle, threads in the same warp can directly share part of their results between registers as soon as they are free. Experimental results show that as the size of the input array increases, the advantages of using shuffle become more obvious. Fig. 11 shows the experimental results. When the problem size reaches 134217728 (2^{27}), using shuffle will increase the execution efficiency by about 25%.

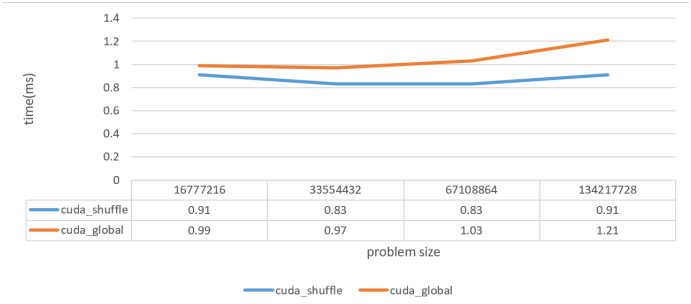


Fig. 11: Performance of reduction using shuffle on NVIDIA Tesla V100 GPUs

F. Bank Conflicts Due to Strided Index

GPU shared memory is architectured into multiple equal-sized memory modules (banks). When shared memory is allocated, consecutive data are sequentially mapped to consecutive 32 banks in cyclic distribution. When different threads in a warp access different locations of the same bank at the same time, access is serialized. This is known as bank conflict, which could significantly impact GPU kernel performance.

We use the classic reduction algorithm as the benchmark to demonstrate the impact of bank conflict on performance. Fig. 12 shows the two kernel functions. The first one is for non-continuous reduction and the size of the stride of each

```
__global__ void sum_bc(REAL *x, REAL *r) {
  __shared__ REAL cache[ThreadsPerBlock];
  int tid=blockIdx.x*blockDim.x+threadIdx.x;
  int cacheId = threadIdx.x;
  cache[cacheId] = x[tid];
  __syncthreads();
  for (int i=1; i<blockDim.x; i*=2) {
    int index = 2*i*cacheId;
    if (index < blockDim.x)
      cache[index]+=cache[index + i];
    __syncthreads();
  }
  if (cacheId == 0)
    r[blockIdx.x] = cache[cacheId];
}

__global__ void sum(REAL *x,REAL *r){
  __shared__ REAL cache[ThreadsPerBlock];
  int tid=blockIdx.x*blockDim.x+threadIdx.x;
  int cacheId = threadIdx.x;
  cache[cacheId] = x[tid];
  __syncthreads();
  for (int i=blockDim.x/2; i>0; i/=2) {
    if (cacheId < i)
      cache[cacheId]+=cache[cacheId+i];
    __syncthreads();
  }
  if (cacheId == 0)
    r[blockIdx.x] = cache[cacheId];
}
```

Fig. 12: Reduction kernels with and without bank conflict

iteration is multiplied by 2, which causes a two-way bank conflict. The second iteration has a stride of 4, which causes a four-way bank conflict. When the stride size reaches 32, all threads access bank0 and the accesses are serialized, causing lowest memory access efficiency. The second kernel function is for continuous reduction. This method has a one-to-one mapping between the thread and the data item, causing no bank conflict. The performance of the two kernels are shown in Fig. 13. As the array size increases, the algorithm's advantages without bank conflicts become more and more apparent.

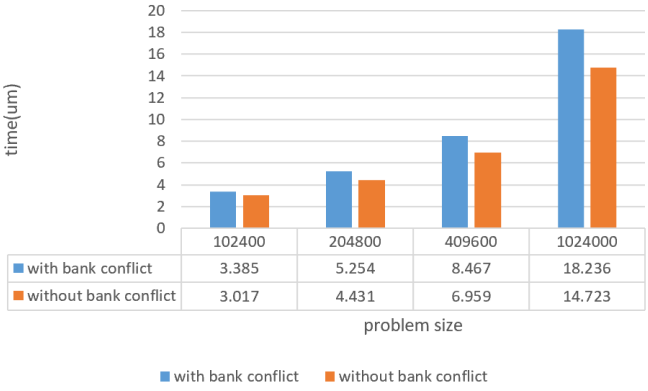


Fig. 13: Performance of reduction kernels with and without bank conflict on NVIDIA Tesla V100 GPUs

V. Properly Arranging Data Movement Between CPU and GPU

In this subsection, we describe microbenchmarks for performance optimization according to the third guideline, which is to properly arrange data movement between CPU and GPU. We demonstrate the challenges of data moving between CPU and GPU using microbenchmarks and the optimized version of the microbenchmarks are included that addresses the challenges.

A. Overlapping and Pipelining GPU Data Copy and Kernel Computation Using CUDA Stream and CudaMemcpyAsync

Data movement between CPU and GPU often dominates the total time of offloading a computation kernel to a GPU. A well-known solution is to use CUDA stream and *cudaMemcpyAsync* to move data between CPU and GPUs. This approach enables parallelism and overlapping between data movement and kernel computation, thus would be able to decrease the impact of data movement to the overall offloading performance. However, the benefit of this approach varies from application to application as the overlapping depends on both the computation kernels and the quantity of data movement. As shown in Fig. 14, for the modified AXPY, using asynchronous copy operations provides a small improvement to performance. Considering AXPY has 1:1 ratio between data movement and computation, data movement is the dominating factor for the performance. Thus even with overlapping, its benefit is minimal. This benchmark is included in the CUDAMicroBench to demonstrate how to use CUDA stream and *cudaMemcpyAsync*. For computation that are likely computation intensive, the benefits would be more obvious.

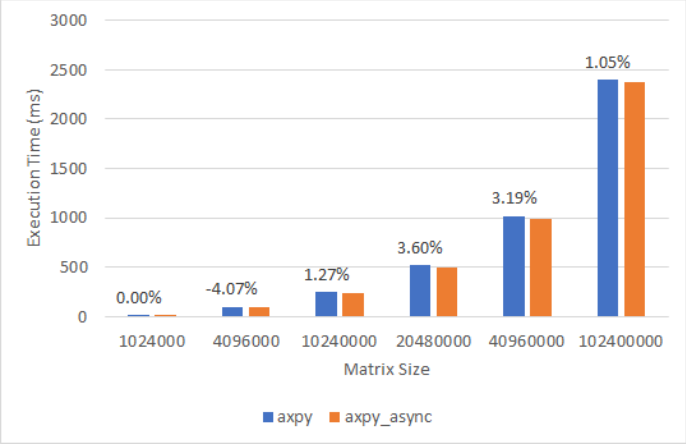


Fig. 14: Performance of asynchronous overlapping of CPU and GPU computation on NVIDIA Tesla V100 GPUs

B. Storing Read-only Data in Read-only Memory

NVIDIA GPUs have reserved part of the DRAM as read-only memory, known as constant memory and texture memory. Properly use read-only memory and global memory by storing read-only data in constant or texture memory and read-write data in global memory improve the usage of the bandwidth of DRAM. The microbenchmark we developed to evaluate the use of the two kinds of read-only memory is matrix addition, which is simple but also allows for evaluating the effect of 2-dimensional texture memory.

The results on the NVIDIA Tesla K80 are shown in Fig. 15. When using texture memory instead of global memory on two matrices of size 20480*20480 we saw a significant performance gap with up 4x the speed. There was not a significant performance difference when using the NVIDIA Tesla V100 instead. This is mainly because of the architecture change in the texture memory unit. Kepler (Tesla K80) has a separate texture cache unit, while Volta (Tesla V100) has the texture cache unit shared with L1 [4].

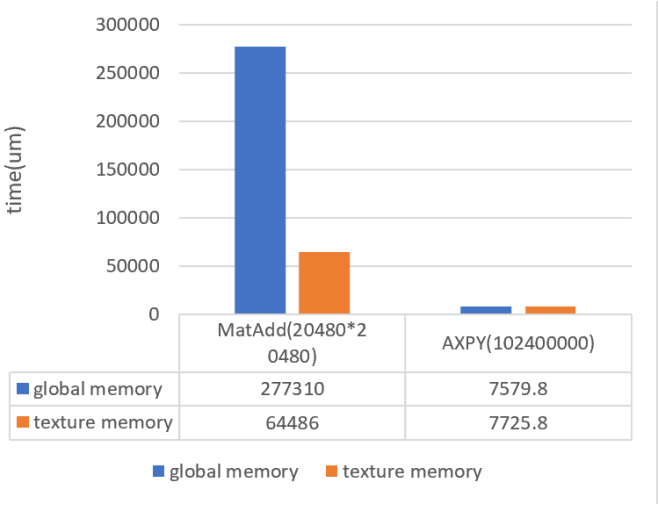


Fig. 15: 1D and 2D texture memory performance on NVIDIA K80 GPUs

C. The Impact of Memory Access Density to Performance

Memory access density refers to the ratio of sizes of the data used for calculation and of the data transferred between CPU and GPUs. Ideally, we would like all data transferred are useful for computation, meaning high memory access density. when the access density is low, moving useless data will impact the performance. One of the optimized solution is to use unified memory such that data are copied from CPU memory to GPU memory when they are needed and accessed. GPU unified memory transfers only the necessary pages which contain the necessary data between CPU and GPU during execution.

Our benchmark to evaluate the density impact is to use stride to control density for AXPY, the larger of the stride, the lower the access density. The performance result shown in Fig. 16 confirms that when the density is low (stride is high), using unified memory significantly improves the performance.

D. Reducing Unnecessary Data Transfer

A simple and classic example of unnecessary data transfer is sparse matrix processing. For a known sparse matrix,

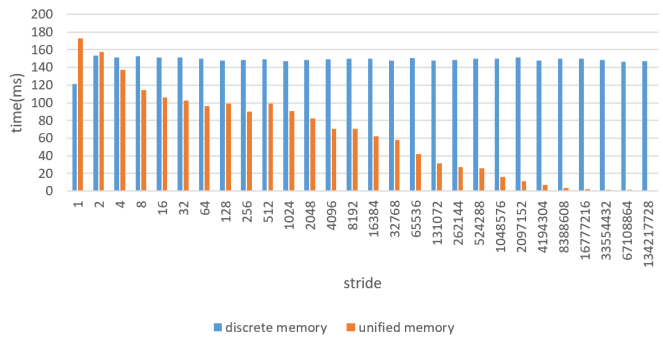


Fig. 16: Performance to show the impact of memory access density on NVIDIA Tesla V100 GPUs

compressed storage format can be used to reduce the amount of data to be transferred and the amount of computation.

We use the example of sparse matrix-vector multiplication (SpMV) to illustrate this problem. The input matrix is sparse and it can be stored in CSR format. When using the traditional row-first storage format and save all elements of the sparse matrix, we need to transfer the entire n * n matrix to the GPU. When using CSR mode to store the matrix, we only need to copy three one-dimensional vectors from the CPU.

The performance of the experiments is shown in Fig. 17. We

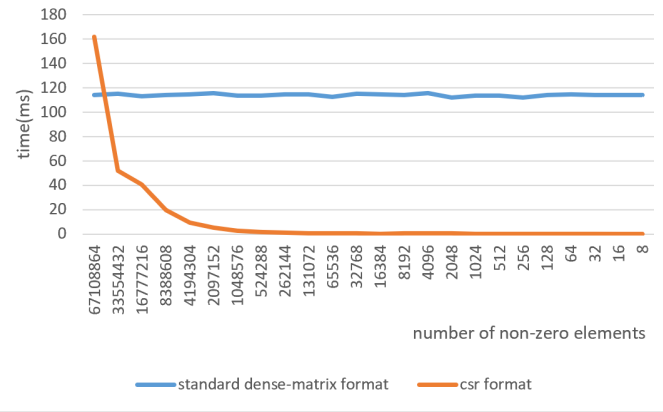


Fig. 17: Performance of SpMV in standard dense-matrix format and CSR format on NVIDIA Tesla V100 GPUs

use a sparse matrix with a size of 10240 * 10240 as the input matrix and change the number of non-zero elements to test two kernels. As the number of non-zero elements decreases, the matrix becomes more sparse, and the advantages of using the CSR format become more obvious.

‘

VI. Related Work

A. Benchmark Suites for Evaluating GPUs

Rodinia is a benchmark suite proposed by Shuai Che et al. for heterogeneous computing [5]. CUDA and OpenMP are used in Rodinia to explore multi-core CPUs and GPUs. Rodinia is structured to span a range of parallelism and data sharing characteristics and can represent different types of behavior according to the Berkeley dwarves. Its significance is to provide the benchmarks for testing the performance of multi-core CPUs and GPUs and get some important architectural insights through the experimental results. The Standard Performance Evaluation Corporation(SPEC), is a company specializing in benchmarks. It includes a SPEC ACCELwhich has a set of compute-intensive parallel applications running under the OpenCL 1.1, OpenACC 1.0, and OpenMP 4.5 APIs. This suite comprehensively evaluates CPU and GPU performance, memory performance, and even compilers' performance. Using CUDA, M Abdullah Shahneous Bari et al. have evaluated a suite of benchmarks including Ray-tracing, Matrix-Matrix Multiplication, Heat Transfer, sparse matrix-vector multiplication in different generations of NVIDIA GPUs, including Kepler, Maxwell, Pascal, and Volta [4]. The primary evaluation is for different GPUs, the performance difference of data placed in different memory including shared memory, constant memory, texture memory, etc.

Unlike the benchmarks mentioned above, the CUDAMicroBench uses much simpler kernels to demonstrate the performance challenges and optimization techniques when using GPUs and CUDA. Thus it fits better to use for performance optimization than those comprehensive and large-application benchmarks.

B. Related Work that Uses Specific Techniques and Features

1) Related work for using CUDA kernel optimization techniques: There are several works that provide insight into kernel optimization techniques. Meng et al. propose compiler side optimizations to dynamically handle warp divergence [6]. This is in contrast to our approach of optimization through code improvements instead of compiler. The work proposed by J DiMarco et al [7] optimize K-means clustering and hierarchical clustering using dynamic parallelism. S Shekofteh et al. studied how to implement concurrent kernel in a more efficient manor including a framework for scheduling [8]. The final work of note is L Toldeo et al. which study the performance impact of task graphs in scheduling, focusing on leveraging concurrent kernel and dynamic parallelism as the kernels to be scheduled with a task graph [9]. The authors claim a 25x performance uplift when using task graphs which illustrates how combining multiple features can lead to significant performance improvement.

2) Related work for using GPU memory: Effective use of different kinds of memory in GPU is a well-studied topic. Zhiyi Yang et al. used shared memory to improve the data reading speed of the DCT encode and decode algorithm in image processing [10]. Paulius Micikevicius et al. optimizes the stencil with shared memory to significantly increase the speed of 3D finite difference computation [11]. The caching algorithm for CUDA shared memory for 2D Smoothed particle hydrodynamics (SPH) solver implementation proposed by Daniel Winkler et al. can significantly improve the efficiency of the original algorithm [12]. Lucas C. G. G. Persoon et al. use texture memory to significantly reduce the calculation

time of γ evaluations without reducing the accuracy [13]. This is a successful application of texture memory in quantify differences in dose distributions. D.P. Playne applied texture memory to three-dimensional Cahn-Hilliard simulations [14]. The unique design of texture memory poses a great advantage in the processing of three-dimensional data. Y Kim et al. proposed CuMAPz to compare the memory performance of CUDA programs [15]. For some effects including data reuse, global memory access coalescing, shared memory bank conflict in shared memory, they explained how CuMAPz optimizes memory strategy.

3) Related work for optimizing CPU-GPU memory copy and overlapping : Using unified memory can optimize CPU-GPU memory copy and overlapping by reducing unnecessary data transmission. R Landaverde et al. developed multiple microbenchmarks for the GPU architecture and tested the performance of Unified Memory Access(UMA) [16]. W Li [17] used the Diffusion3D benchmark in the CUDA SDK samples, Parboil benchmark suite and matrix multiplication to evaluate unified memory technology [18].

CudaMemcpyAsync enables non-blocking data movement between CPU and GPUs. Paulius Micikevicius et al. used *cudaMemcpyAsync* in their work to accelerate 3D Finite Difference Computation [11]. For smaller data sets, communication overhead is close to or even greater than the computation time. Using *cudaMemcpyAsync* can significantly improve the efficiency of program execution. Mohammed Sourouri et al. developed a program for stencil computations based on CPU and GPU [19]. They used *cudaMemcpyAsync* to perform the final CPU-GPU data exchange to achieve a good overlap of various computing activities.

VII. Conclusion

In this paper, we present our development of a microbenchmark suite for assisting users to optimize CUDA programs for NVIDIA GPUs. Each benchmark has kernels for demonstrating performance problems and reference solutions and optimization techniques to address the problem. The microbenchmark can be used to evaluate the performance of CUDA code with different GPU architectures, for validating and comparing software tools for their performance analysis capability, helping users understand the complexity of heterogeneous GPU systems and guiding users to optimize performance.

For future work, we will update and upgrade the benchmark to evaluate new features available in the latest CUDA programming model, e.g. using memory advises to optimize the performance of unified memory. More benchmarks and programming optimization techniques will be added as we improve the study. We will use these microbenchmarks with performance tools and compiler analysis for the purpose of evaluating tools' capability of detecting memory problems. We also plan to develop similar set of benchmarks using OpenCL to evaluate GPUs from other vendors.

Acknowledgment

Funding for this research and development was provided by the National Science Foundation under award number CISE SHF-1551182 and CISE SHF-2015254. The development is also funded by LLNL under Contract DE-AC52-07NA27344 and LLNL-LDRD Program under project 18-ERD-006 (LLNL-CONF-819919).

References

[1] "Adaptive Parallel Computation with CUDA Dynamic Parallelism." [Online]. Available: https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

[2] "CUDA Samples." [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[3] A. Wang, X. Yi, and Y. Yan, "Supporting data shuffle between threads in openmp," in *International Workshop on OpenMP*. Springer, 2020, pp. 98–112.

[4] M. Bari, L. Stoltzfus, P. Lin, C. Liao, M. Emani, and B. Chapman, "Is data placement optimization still relevant on newer gpus?" Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[6] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 235–246.

[7] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," 05 2013, p. 87520E.

[8] S. . Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning, and H. S. Yazdi, "ccuda: Effective co-scheduling of concurrent kernels on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 766–778, 2020.

[9] L. Toledo, A. J. Peña, S. Catalán, and P. Valero-Lara, "Tasking in accelerators: Performance evaluation," in *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2019, pp. 127–132.

[10] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *2008 International Conference on Computer Science and Software Engineering*, vol. 3. IEEE, 2008, pp. 198–201.

[11] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, 2009, pp. 79–84.

[12] D. Winkler, M. Meister, M. Rezavand, and W. Rauch, "gpusphase—a shared memory caching implementation for 2d sph using cuda," *Computer Physics Communications*, vol. 213, pp. 165–180, 2017.

[13] L. C. Persoon, M. Podesta, W. J. van Elmpt, S. M. Nijsten, and F. Verhaegen, "A fast three-dimensional gamma evaluation using a gpu utilizing texture memory for on-the-fly interpolations," *Medical physics*, vol. 38, no. 7, pp. 4032–4035, 2011.

[14] D. P. Playne and K. A. Hawick, "Data parallel three-dimensional cahn-hilliard field equation simulation on gpus with cuda." in *PDPTA*, vol. 9, 2009, pp. 104–110.

[15] Y. Kim and A. Shrivastava, "Cumapz: A tool to analyze memory access patterns in cuda," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 128–133.

[16] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[17] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 1092–1098.

[18] D. Coombes, "Tegra k1 whitepaper," Tech. rep, NVIDIA, Tech. Rep., 2014.

[19] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden, and X. Cai, "Cpu+ gpu programming of stencil computations for resource-efficient use of gpu clusters," in *2015 IEEE 18th International Conference on Computational Science and Engineering*. IEEE, 2015, pp. 17–26.