Evaluating CUDA-Aware Approximate Computing Techniques

Işıl Öz[1]

[1]Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey

Abstract

Approximate computing techniques offer performance improvements by performing inexact computations. Moreover, CUDA programs written to be executed on GPU devices employ specific features to utilize the parallel computation units of heterogeneous GPU architectures. While generic software-level approximate computing techniques have been applied to heterogeneous CUDA programs, CUDA-specific approaches may introduce promising performance improvements by not corrupting the target computations. In this work, we propose software approximation techniques for CUDA programs: kernel-aware loop perforation, partition-level synchronization, block-level atomic operations, and warp divergence elimination. We perform source code transformations on target benchmark programs by applying our techniques. We evaluate performance improvements by trading off accuracy in our target computations. Our experimental results reveal that CUDA-aware approximation techniques offer significant performance improvements at the expense of acceptable accuracy loss.

Keywords

CUDA programming model, GPU computing, approximate computing

1. Introduction

Heterogeneous computer systems, combining general-purpose processors and GPU devices as accelerators, enable high-performance and energy-efficient executions. However, the applications from various domains such as AI acceleration, big-data processing, and high-performance computing (HPC) with large computing requirements make power consumption an important concern in these systems [1]. Since modern GPU architectures employ complex structures and the target workloads exploit the massively parallel resources, energy efficiency becomes critical for large-scale GPU executions [2, 3, 4].

To solve the conflict between performance and energy efficiency, approximate computing maintains high performance and low power consumption for applications that can tolerate inexact computations. While the architecture-level approximate computing techniques are enabled by modifying processor units and memory components, software solutions based on compiler transformations or manual code modifications also offer approximate computations [5, 6]. While using inexact hardware or voltage scaling maintains hardware solutions, techniques like loop perforation or relaxed synchronization offer performance-accuracy tradeoffs at the software level.

Since GPU systems aim for applications from different domains, they employ approximate computing techniques to improve performance and energy efficiency by trading with the inaccuracy in the target computations. Besides inherently error-tolerant graphics and image processing applications [7], general-purpose GPU programs benefit from approximations with reasonable incorrect computations [8]. While some works reuse generic techniques like perforation [9], some methods utilize GPU-specific hardware or software components to employ approximations [10]. Additionally, simulation-based evaluations propose hardware modifications by either approximate units or supporting the approximate computations [11, 12].

In this work, we propose software-based approximations for CUDA programs running on GPU architectures. Not only do we adapt the existing techniques for the GPU programs, but we also propose CUDA-specific methods to target parallel CUDA threads. Our main contributions are as follows:

- We propose kernel-aware loop perforation by adapting the loop perforation technique for CUDA programs. To reduce the synchronization overhead, we propose partition-level synchronization and block-level atomic operations based on CUDA cooperative groups and CUDA thread scope atomic functions. Additionally, we eliminate the warp divergence inside CUDA kernel functions to prevent serial execution caused by branch instructions.
- We modify the target codes by inserting compiler directives enabling our techniques, and generate our approximate versions based on the given compiler options.
- We perform an experimental study to evaluate the impact of the modifications by our approximations. Our experimental study includes applications from different domains to observe the performance and accuracy variations for the target execution. Our experimental results reveal that CUDA-aware approximation techniques offer significant performance improvements at the expense of acceptable accuracy loss.

The remainder of this paper is organized as follows: Section 2 presents some background on approximate computing and the CUDA programming model. We explain our approximation methods in Section 3. Then, the experimental results are outlined in Section 4. Section 5 presents relevant studies about CUDA approximations. Finally, in Section 6, we summarize the work with some conclusive remarks.

2. Background and Motivation

2.1. Approximate Computing

Approximate computing introduces acceptable inaccuracies into the computing process and promises significant performance and energy gains. Some techniques employ the loop perforation approach, which works by skipping some loop iterations to reduce computational overhead [9, 13]. Relaxed synchronization shortens the waiting time of the threads that wait for the completion of the other threads' work

RAW'24: The 3rd workshop on Resource AWareness of Systems and Society (RAW 2024), July 02–05, 2024, Maribor, Slovenia

isiloz@iyte.edu.tr (I. Öz)

0000-0002-8310-1143 (I. Öz)

```
__global__ void vectorAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    if (id < n)
        c[id] = a[id] + b[id];
}

int main(int argc, char** argv)
{
    double *d_a, d_b, d_c;
    int n;

    //... memory allocations and copy operations

    int blockSize = 1024;// Number of threads in each thread block
    int gridSize =  (int)ceil((float)n/blockSize);

    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    //... memory copy and deallocations
}
```

Listing 1: Vector addition kernel function and its launch in CUDA.

[14, 15]. Reduced precision computation employs fewer compute cycles with insignificant value modifications for error-tolerant applications targeting low-precision executions [16].

2.2. GPU Programming Model

While modern GPU architectures evolve as the requirements of the target applications change, GPU devices employ SIMT (Single Instruction Multiple Threads) execution to accelerate data processing tasks in their parallel execution cores.

A program written in the CUDA programming model [17], which is a parallel programming model for NVIDIA GPU devices, starts its execution in a CPU, allocates memory space on the GPU, transfers data into GPU global memory, and starts a kernel function execution by creating thousands of threads. Each thread executes the same program (SIMT) by processing different parts of the given data. Threads that execute on the GPU are part of a compute kernel specified by a function. Besides data-parallel applications that can benefit from many parallel execution units of GPUs, large-scale irregular computations utilize the massive degree of parallelism and the high memory bandwidth provided by GPUs. Listing 1 presents the CUDA kernel function and kernel launch configuration for the vector addition operation. For simplicity, we skip the memory allocation and copy operations. The execution launches the *vectorAdd* function by specifying the number of blocks and the number of threads in each block. The hardware scheduler schedules the blocks into SM (Streaming Multiprocessor) units and thread groups (warps) into GPU cores inside SMs. Each thread executes the kernel function and performs the addition operation based on its global thread identifier.

3. Approximation for CUDA Programs

We propose three main approximations for target CUDA programs. Firstly, we exploit loop perforation by adapting the popular method for loop and loop-similar structures in the CUDA code. Secondly, we replace synchronization primitives with relaxed versions and propose partition-level synchronization for threads based on CUDA cooperative groups and block-level atomic operations using CUDA thread scopes. Finally, we remove the warp divergence, a serial bottleneck in GPU executions.

We evaluate target CUDA source codes and perform code transformations based on compiler directives. While our approach requires manual code analysis and modifications to introduce directives in code segments, the semi-automatic configuration enables us to generate target codes that employ approximations selectively by compiling the code with specific options.

3.1. Kernel-Aware Loop Perforation

While loop perforation skips some loop iterations in a serial program, the same technique can be applied to CUDA programs by adapting the perforation accordingly. We consider three approximation versions derived from loop perforation:

Kernel launch perforation: We skip the iterations of the loop, which launches one or multiple CUDA kernel functions at each iteration. The perforation is simply a regular loop perforation with kernel launches per iteration. In a code segment given in Listing 2 (*Fdtd2d* program from *Polybench* suite [18]), we reduce the number of iterations and consequently, the kernel launches by assigning a smaller upper bound (for *_PB_TMAX* variable in the example code).

Kernel launch configuration perforation: In data-parallel CUDA programs, CUDA threads execute implicit loops in parallel by performing the computations that belong to one or more iterations of the serial program loop. We launch target kernel executions by reducing the number of threads in the configuration; hence, the original loop in the serial program is perforated. In a code segment given in Listing 3 (*Convolution2D* program from *Polybench* suite [18]), we modify the *block* or *grid* configuration parameters of the *convolution2D_kernel* kernel by reducing the X or Y dimensions of the grid.

```
...
for(int t = 0; t < _PB_TMAX; t++)
{
    fdtd_step1_kernel<<<grid,block>>>(nx, ny, _fict_gpu, ex_gpu, ey_gpu, hz_gpu, t);
    cudaDeviceSynchronize();
    fdtd_step2_kernel<<<grid,block>>>(nx, ny, ex_gpu, ey_gpu, hz_gpu, t);
    cudaDeviceSynchronize();
    fdtd_step3_kernel<<<grid,block>>>(nx, ny, ex_gpu, ey_gpu, hz_gpu, t);
    cudaDeviceSynchronize();
}
...
```

Listing 2: The *Fdtd2d* code with kernel function calls inside a loop.

```
...
dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
dim3 grid(ceil(((float)NI) / ((float)block.x)), ceil(((float)NJ) / ((float)block.y)));

convolution2D_kernel <<< grid,block >>> (ni, nj, A_gpu,B_gpu);
...
```

Listing 3: The *Convolution2D* code kernel launch configuration.

```
...
__global__ void mean_kernel(int m, int n, DATA_TYPE *
     mean, DATA_TYPE *data)
{
 int j = blockIdx.x * blockDim.x + threadIdx.x;

 if (j < _PB_M)
 {
    mean[j] = 0.0;

    int i;
    for(i = 0; i < _PB_N; i++)
    {
        mean[j] += data[i * M + j];
    }
    mean[j] /= (DATA_TYPE)FLOAT_N;
 }
}
...
```

Listing 4: The *Covariance* code with loop structures inside a kernel function *mean_kernel*.

Intra-kernel loop perforation: We perform the standard loop perforation method for the code inside kernel functions. In a code segment given in Listing 4 (*Covariance* program from *Polybench* suite [18]), we reduce the number of loop iterations inside *mean_kernel* kernel (*_PB_N* variable in the example code).

We modify each program code by inserting compiler directives for a set of loop perforation types. Specifically, we define four directives: *KERNEL_LAUNCH_PERFORATION*, *GRID_PERFORATION*, *BLOCK_PERFORATION*, *LOOP_PERFORATION*, and compile the programs by enabling the directives with specific values, which represent the perforation rate as the reduction ratio of the target loop. By enabling the chosen perforation type(s) and rate(s) at compile time, we evaluate the impacts on the execution.

3.2. Relaxed Synchronization

Multiple CUDA threads require time-consuming synchronization to access shared data or resolve data dependencies, utilizing atomic operations (like *atomicAdd*) and barrier operations (like *__syncthreads()*), respectively. The relaxed synchronization offers performance gains by synchronizing fewer threads in exchange for output accuracy loss. We consider two main relaxations based on CUDA cooperative groups and CUDA thread scopes:

Partition-level synchronization: CUDA threads within a block can cooperate by synchronizing their execution to coordinate memory accesses. The programmer can define synchronization points by calling the *__syncthreads()* function, which acts as a barrier and makes waiting for all threads. While CUDA employs block-level synchronization by *__syncthreads()* function, *__syncwarp()* function, which synchronizes the threads within a warp, has become available on CUDA 9. This is important for porting code to modern GPU architectures after *Volta*, in which threads within a warp can be scheduled separately. Additionally, the Cooperative Groups API [19] provides a rich set of thread-synchronization primitives by forming partitions with a set of threads. Listing 5 presents different code snippets to organize groups of threads. While the first group, *blockgroup*, represents all the threads in a thread block, *warpgroup* represents all the threads in a warp. If we want to synchronize the threads in those groups, the behavior will be the same with *__syncthreads()* and *__syncwarp()* functions, respectively.

For implementing the approximation, first, we search for all *__syncthreads()* function calls in the target kernel functions and configure the synchronization level for each synchronization point. Specifically, we either completely skip *__syncthreads()* (**SKIP**) or replace it with a relaxed version. For relaxing synchronization, we choose *__syncwarp()* (**WARP**) or utilize cooperative thread groups (the details are given below). We modify each program code and inject *#ifdef* directives to guide the compiler based on user preferences. For each *__syncthreads()* code block, we define one directive and compile the code by specifying one or more directives.

```
// Cooperative group for the current thread block
auto blockgroup = cooperative_groups::this_thread_block();

// Cooperative group for each warp in the thread block
auto warpgroup = cooperative_groups::tiled_partition<32>(threadblock);

// Cooperative group for each 16 threads in the thread block
auto subwarp16 = cooperative_groups::tiled_partition<16>(threadblock);

// Cooperative group for all currently coalesced threads in the warp
auto coalescedgroup = cooperative_groups::coalesced_threads();

// Thread block groups can sync
blockgroup.sync();
```

Listing 5: CUDA cooperative groups [19].

```
//Replaced code version 1 (4TILE)
thread_group tile32 = tiled_partition(this_thread_block(), 32);
thread_group tile4 = tiled_partition(tile32, 4);
tile4.sync();

//Replaced code version 2 (ACTIVE)
thread_group active = coalesced_threads();
active.sync();
```

Listing 6: Partition-level synchronization configurations.

For our partition-level approach, we define two thread partitions (as given in Listing 6): 1) **4TILE**: Cooperative thread groups with four threads in the corresponding warp, 2) **ACTIVE**: Currently coalesced threads in the warp. When data-dependent conditional branches in the code cause threads within a warp to diverge, the SM disables (deactivates) threads that do not take the branch. The threads that remain active on the path are referred to as coalesced.

Block-level atomic operations: While the atomic operations in standard C or C++ are uniform, the CUDA programming model offers atomic functions at different scopes. A thread scope specifies the set of threads that can synchronize with each other using atomic operations. Atomic functions with *_system* suffix (e.g., *atomicAdd_system*) are atomic at system scope, where the system refers to the system running on multiple GPUs and CPUs. Atomic functions without a suffix (e.g., *atomicAdd*) are atomic at device scope, where the device refers to the target GPU device. Atomic functions with *_block* suffix (e.g., *atomicAdd_block*) are atomic at thread block scope, which refers to the synchronization of the threads executing on the same thread block.

In block-level atomic operations, we target that the threads perform atomic operations at the largest thread block scope. Like our synchronization approach, we search for all atomic functions in the target CUDA code and reduce the atomic scope accordingly. For instance, we replace *atomicAdd* function calls with *atomicAdd_block*, or we completely remove the function call. Hence, we aim for atomic operations with fewer threads than the original code. Similarly, we could replace *atomicAdd_system* or skip *atomicAdd_block* functions.

3.3. Warp Divergence Elimination

CUDA threads are executed in groups of 32 threads (warps), and all threads in a warp execute the same instruction at the same time. Due to SIMD execution model, when threads in the same warp need to perform different operations, the execution of the different branches is serialized, thus hurting performance improvement that could be gained from parallelism. Figure 1 presents an example scenario for warp divergence. The eight threads (assuming we have an 8-thread warp size for simplicity) start the kernel execution, then at *Branch* point, there is an *if* statement that causes different path executions. While four threads execute the instruction at *Path A*, the other four continue the execution at *Path B*. When the first four threads execute *Path A*, the others must wait and perform no operation. The marked execution prevents full warp utilization by activating only four threads simultaneously in an 8-thread warp structure.

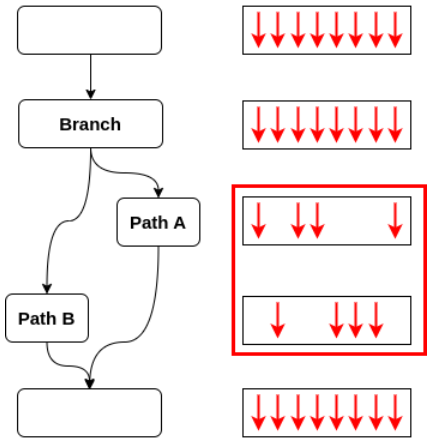

Figure 1: SIMD Warp Divergence [20].

To eliminate the divergence overhead, as an approximation method, we execute only one path in case of multiple paths in a warp. For instance, for the code given in Listing 7 (from *Grappolo* application [21]), we configure to execute

Path 1, *Path 2*, or *Path 3*. Alternatively, we completely skip the divergent code segment in our evaluations.

```
...
if (currCId == FLAG_FREE) {
    /*Path 1*/
}
if (currCId == (1 + dataItem->cId)) {
    /*Path 2*/
}
else
{
    /*Path 3*/
}
...
```

Listing 7: SIMD warp divergence code example.

4. Experimental Study

4.1. Experimental Setup

To evaluate our approximation methods, we select CUDA applications from *Polybench* [18] and *Gardenia* [22] benchmark suites and utilize an optimized CUDA implementation of the Louvain community detection algorithm, namely *Grappolo* [21]. While *Polybench* applications mostly employ data-parallel computations with multiple loop structures, *Gardenia* implements a set of graph algorithms that include synchronization primitives. *Grappolo*, with computationally intensive and complex structures, includes code segments for our evaluations on relaxed synchronization and warp divergence-based approximations.

We compile our programs with CUDA 12.1 [23] and run our approximation experiments in a system with an NVIDIA GeForce RTX 3050 Ti Mobile GPU device. The GPU device, built on Ampere architecture [24], has 4 GB GDDR6 memory.

4.2. Experimental Results

We evaluate our three main approximations for target CUDA programs separately. We execute both original and approximated versions, measure GPU execution times, and collect result outputs. By comparing execution time and output accuracies, we perform a tradeoff analysis for target computations.

4.2.1. Kernel-Aware Loop Perforation

For our loop perforation techniques, we select six programs from the *Polybench* benchmark suite [18]. The programs have data-parallel characteristics and each employs different loop structures. We execute *Correlation*, *Covariance*, *Syrk*, *Fdtd2d* with *STANDARD* input sizes and *Jacobi-2D*, and *2DConv* with *LARGE* input sizes to have longer execution times. We collect GPU execution times and incorrect computations by comparing them with the original output. Since the programs work with array structures and compute array elements as the final output, we evaluate the number of array elements that are computed incorrectly.

Figure 2 demonstrates performance improvement and inaccuracy values for the programs when our loop-perforation methods are applied. For each applicable method, namely, kernel launch perforation, intra-kernel loop perforation, grid-level kernel launch configuration perforation, and block-level kernel launch configuration perforation, we perform 90% and 80% perforation rates. If the program does not support the target approximation (e.g., *Correlation* does not have a kernel launch inside a loop), we simply do not have the corresponding result in our evaluation. The values in Figure 2 present 1/Speedup and the rate of incorrectly computed elements. We define the performance in terms of speedup, the ratio of the compute time for the original execution to the time for the approximate execution, and report the 1/Speedup values in our results. For instance, the execution time for the original *Correlation* execution is 1.785 milliseconds, and it computes 4194304 array elements. When we perforate the kernel function loops by 90% (*Loop (90%)*), we have 1.302 milliseconds and 793356 incorrect computations. Therefore, the performance improvement rate equals 1.302/1.785=**0.73**, and the rate of the incorrect computations is 793356/4194304=**0.19**, shown in Figure 2. By reporting performance improvement and inaccuracy values in this way, one can evaluate performance gains and incorrect results for each approximation and make design decisions. Based on the program characteristics, each approximation affects the execution outcome differently. We can have up to 60% performance improvements (*Loop (80%)* for *Correlation*) in exchange for 40% of the elements incorrectly computed. Some approximations offer good tradeoff points, like grid-level kernel launch configuration perforations (*Grid (90%)* and *Grid (80%)*) in *Fdtd2d*. We can have 20% and 30% performance improvements by losing 30% and 50% of correct computations. On the other hand, there is no performance improvement with small inaccuracy values (like kernel launch configuration perforations in *Covariance*) or intolerable output loss with improvement in execution times (like loop perforations in *Syrk*).

4.2.2. Relaxed Synchronization

We evaluate the *Betweenness Centrality (bc)* in the *Gardenia* benchmark suite [22], which has four different implementations. For a sample graph (*soc-LiveJournal1* [25]), we execute each version and select the one with the lowest execution time. Since the version already employs optimizations, we apply our approximation methods to that version for fair comparison. The implementation (i.e., *bc_topo_lb*) has four main kernel functions with synchronization primitives (i.e., *__syncthreads()*). We apply our relaxed synchronization techniques for each seven *__syncthreads()* function call in the target kernel functions and perform four specific modifications: 1) **SKIP:** Remove *__syncthreads()*, 2) **WARP:** Synchronize threads in the same warp, 3) **4TILE:** Synchronize four threads in the same cooperative group, 4) **ACTIVE:** Synchronize coalesced threads. Finally, we have 28 different versions. We execute those versions with 19 different datasets. We observe execution time and output differences for only a subset of our executions, specifically, relaxations for three *__syncthreads()* function calls on only one kernel function. For three synchronization points, we also consider the relaxation of their combinations.

Table 1 presents the execution times and the number of incorrect computations in the observed output for the specified graphs. We can observe that *SKIP*, *WARP*, and *ACTIVE* mostly outperform *4TILE*, probably due to the overhead of fine-grained group creation. While relaxing individual synchronization points (i.e., *SYNC 1*, *SYNC 2*, *SYNC 3*) offers performance gains significantly with non-significant accu-

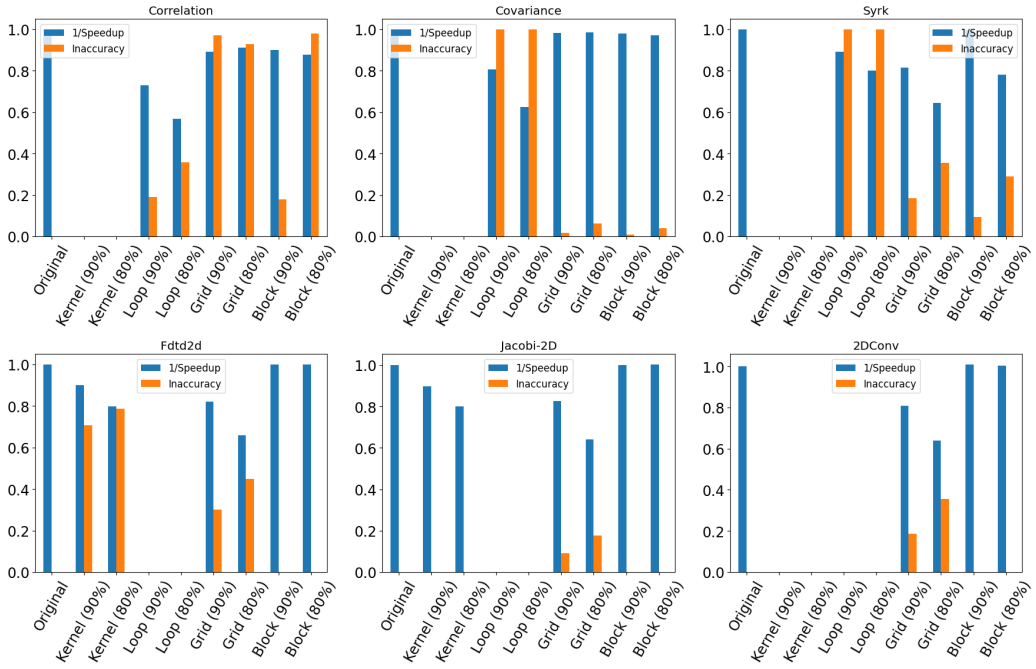

Figure 2: Speedup-Inaccuracy variation for loop perforation approximation methods.

Table 1
Execution time and incorrect computations (out of given expected correct values) for relaxed synchronization methods for *bc* application.

		ljournal-2008		**socLiveJournal**		**cage15**	
		Time	Incorrect (5,363,260)	Time	Incorrect (4,847,571)	Time	Incorrect (5,154,859)
ORIGINAL		61.156	0	57.116	0	47.096	0
SYNC 1	SKIP	60.637	954	57.116	276	46.780	28,326
	WARP	60.199	793	53.611	355	47.109	70,611
	4TILE	60.579	754	57.205	227	46.939	76,002
	ACTIVE	60.205	1113	55.431	292	47.164	71,563
SYNC 2	SKIP	59.770	19,888	52.756	1636	47.425	486,970
	WARP	59.314	11,230	52.485	1693	47.811	498,751
	4TILE	61.364	14,087	52.300	1522	48.465	303,306
	ACTIVE	59.397	8345	52.532	1647	47.850	463,191
SYNC 3	SKIP	61.328	64	53.996	36	47.198	15,531
	WARP	60.643	1996	53.774	38	47.843	14,341
	4TILE	61.374	137	53.961	34	47.186	13,383
	ACTIVE	60.744	160	53.872	32	47.775	13,975
SYNC 1+2	SKIP	**56.800**	20,137	**50.572**	1739	**46.420**	98,691
	WARP	**56.377**	17,035	**50.401**	1633	**46.879**	104,232
	4TILE	57.779	12,311	51.199	1678	47.217	138,951
	ACTIVE	**56.195**	22,491	**50.495**	1657	**46.908**	84,210
SYNC 1+2+3	SKIP	57.088	20,346	50.871	1729	46.966	85,029
	WARP	56.747	20,719	50.774	1600	47.320	62,026
	4TILE	57.801	14,807	51.513	1649	47.678	59,262
	ACTIVE	56.608	17,248	50.880	1556	47.352	87,300

racy loss, the combinations (i.e., *SYNC 1+2*, *SYNC 1+2+3*) further improve the performance without hurting the output quality much. While the accuracy loss depends on the target dataset, we see the most promising relaxation options for performance gains (around 8%-10%) with the *SYNC 1+2* version.

Since *bc* has no atomic operations, we consider another application to observe the impact of our approximation techniques for atomic operations. We utilize *Grappolo* code [21], a highly-optimized CUDA implementation of the Louvain community detection algorithm [26]. Louvain is a greedy graph processing method that assigns each vertex to a community, which maximizes the overall *Modularity* and generates a new graph in which the communities become new vertices. Since the output metric, *Modularity*, does not present an exact result, trading the output accuracy with performance improvement can be an interesting evaluation for the execution. The *Modularity* metric evaluation depends on the application domain utilizing the community detection, however, a *Modularity* value close to 1 presents

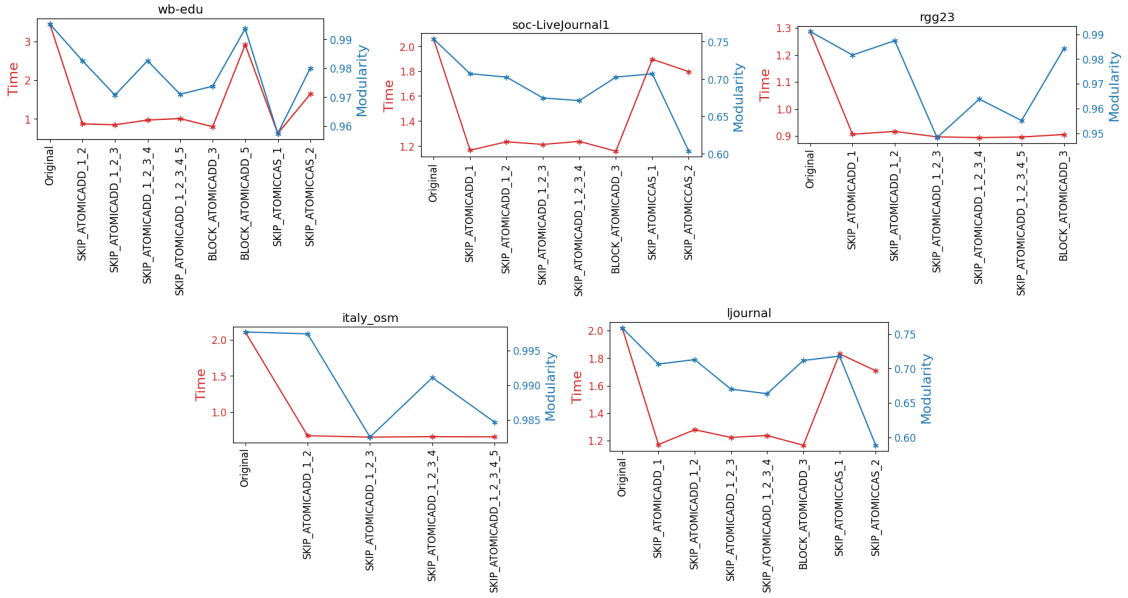

Figure 3: Execution time-Modularity values for relaxing atomic operations in *Grappolo*.

higher quality output. While the *Grappolo* employs both synchronization and atomic operation primitives, we evaluate only atomic operations and perform our relaxation methods. Specifically, for *atomicAdd* and *atomicCAS* function calls, we either replace them with the non-atomic operation or the block-level atomic function calls (i.e., *atomicAdd_block* or *atomicCAS_block*).

Figure 3 presents execution time and modularity values as the performance and the accuracy metrics, respectively. Besides individual atomic operations, we relax the combinations of the atomic operations to see the impact on the outcome. In our target program, we have five *atomicAdd* and three *atomicCAS* function calls. We label the relaxations by considering the index and type of the method. Specifically, we use *SKIP* or *BLOCK* as the prefix and the order of the corresponding function as the suffix. For instance, *SKIP_ATOMICADD_1* replaces the first *atomicAdd* with the non-atomic operation; *BLOCK_ATOMICADD_3* replaces the third *atomicAdd* with *atomicAdd_block*. For the combined relaxations, we concatenate the index of each operation such as *SKIP_ATOMICADD_1_2*. We only select a subset of the combinations since it is not practical to execute all of them. While we conduct experiments for 19 datasets, we include five of them that present the most interesting design points. All five graphs demonstrate significant performance improvements with little modularity losses. Depending on the modularity evaluation of the target domain utilizing the community detection, one can easily prefer approximated versions. The executions that have large modularity values in the original version exhibit significant performance improvements without hurting the modularity very much. Especially, *SKIP_ATOMIC_ADD_1_2* version promises up to *3x* performance gains with *0.01* modularity loss.

4.2.3. Warp Divergence Elimination

We utilize *Grappolo* for our warp divergence elimination method due to its complex structure that employs branch instructions inside kernel functions. We work with two kernel functions and perform different divergence elimination. Firstly, we execute only one path out of three, but our execution does not end (infinite loop) with each path choice. Then, we apply a different strategy by eliminating the code in the target path executions and returning the previously computed value with no computation. For this method, our execution is completed in a shorter time with lower Modularity values.

Table 2 presents execution time and modularity values for the original execution and our approximate version. While we can see a decrease in all execution times, the approximation also destroys modularity values with one exception, namely the *wb-edu* dataset. Since this approximation completely eliminates some code segments, accuracy loss becomes inevitable for most cases, and it requires a more rigorous analysis of the target code.

5. Related Work

SAGE [7] presents a static compiler that generates a set of approximated CUDA kernels and a runtime system that employs selective discarding of atomic operations, data packing, and thread fusion optimizations. It yields 2.5× speedup with less than 10% quality loss for machine learning and image processing kernels. While SAGE proposes approximations for CUDA computations and significantly improves performance, it relies on generic approximation methods instead of CUDA-specific techniques.

Freytag et al. [27] propose efficient executions for scientific simulation applications by building multiple kernel implementations with different precision levels. They execute approximated kernel versions by switching from one version to another at runtime based on Target Output Quality (TOQ) scenarios. By employing execution configurations based on an analysis of the accuracy loss, the experiments reveal high-performance and energy-efficient executions for target precision levels. While the authors build application-layer approximations for the target code, they modify the precision levels of the target code while not introducing CUDA-specific methods.

Table 2
Execution time and Modularity values with warp divergence elimination for *Grappolo*.

	Original		Approx.	
Dataset	Time	Modularity	Time	Modularity
relat9	1.206	0.491	0.616	0.254
cage15	1.341	0.893	1.077	0.727
rel9	1.094	0.458	0.446	0.253
ljournal	2.034	0.759	1.573	0.588
rgg23	1.295	0.991	1.206	0.718
soc-LiveJournal1	2.042	0.753	1.606	0.603
wb-edu	3.468	0.995	1.621	0.980

Liu et al. [28] present cuSpAMM, the CUDA adaptation of the Sparse Approximate Matrix Multiply algorithm, by utilizing thread parallelism, memory tiling, and the tensor cores in multiple GPU devices. While the proposed work implements an approximation algorithm by considering GPU-specific features, the implementation, rather than approximation, relies on GPU optimization techniques.

6. Conclusions and Future Work

In this work, we propose CUDA-specific approximation methods based on loop perforation, relaxed synchronization, and warp divergence elimination. We define approximations as compiler directives and enable them for target executions. Our experimental results demonstrate that our approximation techniques promise good performance improvements without hurting output accuracy significantly.

Our approximations are enabled based on compiler directives. While the directives offer some level of automation, we can extend our work by building a fully automated tool that performs source-to-source compiler transformations. Thus, we can easily generate our approximated versions. Moreover, a design space exploration technique potentially helps to choose the best design points considering performance improvements and inaccuracy values.

While approximate computing offers performance improvements, it is essential to evaluate the power consumption of the target execution. We can extend our work by including energy measurements for GPU devices and include that criterion as our resource-aware evaluation.

Acknowledgments

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Grant No: 122E395. This work is partially supported by CERCIRAS COST Action CA19135 funded by COST Association.

References

[1] S. Mittal, J. S. Vetter, A survey of methods for analyzing and improving GPU energy efficiency, ACM Computing Surveys 47 (2014). URL: https://doi.org/10.1145/2636342. doi:10.1145/2636342.

[2] K. Ma, X. Li, W. Chen, C. Zhang, X. Wang, GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures, in: 2012 41st International Conference on Parallel Processing, 2012, pp. 48–57.

[3] Q. Zeng, Y. Du, K. Huang, K. K. Leung, Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing, IEEE Transactions on Wireless Communications 20 (2021) 7947–7962.

[4] V. Raca, S. W. Umboh, E. Mehofer, B. Scholz, Runtime and energy constrained work scheduling for heterogeneous systems, The Journal of Supercomputing 78 (2022) 17150–17177.

[5] S. Mittal, A survey of techniques for approximate computing, ACM Computing Surveys 48 (2016).

[6] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, D. Zufferey, Exploiting errors for efficiency: A survey from circuits to applications, ACM Computing Surveys 53 (2020). doi:10.1145/3394898.

[7] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, S. Mahlke, Sage: Self-tuning approximation for graphics engines, in: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[8] D. Peroni, M. Imani, H. Nejatollahi, N. Dutt, T. Rosing, Arga: Approximate reuse for GPGPU acceleration, in: 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1–6.

[9] D. Maier, B. Cosenza, B. Juurlink, Local memory-aware kernel perforation, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 278–287. doi:10.1145/3168814.

[10] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, H. Corporaal, Sfu-driven transparent approximation acceleration on GPUs, in: Proceedings of the 2016 International Conference on Supercomputing, ICS '16, Association for Computing Machinery, New York, NY, USA, 2016.

[11] R. Garcia, F. Asgarinejad, B. Khaleghi, T. Rosing, M. Imani, Trulook: A framework for configurable GPU approximation, in: 2021 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2021, pp. 487–490. doi:10.23919/DATE51398.2021.9474239.

[12] F. Vaverka, V. Mrazek, Z. Vasicek, L. Sekanina, Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on GPU, in: 2020 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2020, pp. 294–297. doi:10.23919/DATE48585.2020.9116299.

[13] D. Maier, B. Juurlink, Model-based loop perforation,

in: Euro-Par 2021: Parallel Processing Workshops, Springer International Publishing, Cham, 2022, pp. 549–554.

[14] K. Iliakis, H. Timko, S. Xydis, P. Tsapatsaris, D. Soudris, Enabling large scale simulations for particle accelerators, IEEE Transactions on Parallel and Distributed Systems 33 (2022) 4425–4439. doi:10.1109/TPDS.2022.3192707.

[15] A. L. C. Bueno, N. de La Rocque Rodriguez, E. D. Sotelino, Adaptive relaxed synchronization through the use of supervised learning methods, Future Generation Computer Systems 106 (2020) 260–269.

[16] S. Cherubin, G. Agosta, Tools for reduced precision computation: A survey, ACM Computing Surveys 53 (2020). URL: https://doi.org/10.1145/3381039. doi:10.1145/3381039.

[17] D. B. Kirk, W. mei W. Hwu, Programming Massively Parallel Processors (Third Edition), Morgan Kaufmann, 2017.

[18] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, 2012 Innovative Parallel Computing (InPar), 2012.

[19] M. Harris, K. Perelygin, Cooperative groups: Flexible CUDA thread programming, https://developer.nvidia.com/blog/cooperative-groups/, 2017.

[20] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, M. Martonosi, General-Purpose Graphics Processor Architecture, 2018.

[21] A. K. Mahantesh Halappanavar, Howard (Hao) Lu, S. Ghosh, Grappolo community detection, https://github.com/ECP-ExaGraph/grappolo, 2024.

[22] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, C. Yang, Gardenia: A graph processing benchmark suite for next-generation accelerators, ACM Journal on Emerging Technologies in Computing Systems 15 (2019). URL: https://doi.org/10.1145/3283450. doi:10.1145/3283450.

[23] Nvidia, CUDA toolkit 12.1, https://developer.nvidia.com/cuda-12-1-0-download-archive, 2024.

[24] Nvidia, Nvidia ampere ga102 GPU architecture white paper, https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf, 2021.

[25] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.

[26] V. Blondel, J. Guillaume, R. Lambiotte, E. Mech, Fast unfolding of communities in large networks, Journal of Statistical Mechanics: Theory and Experiment (2008).

[27] G. Freytag, C. A. Künas, P. Rech, P. O. A. Navaux, Interleaved execution of approximated cuda kernels in iterative applications, in: 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2024, pp. 60–67. doi:10.1109/PDP62718.2024.00017.

[28] X. Liu, Y. Liu, H. Yang, M. Dun, B. Yin, Z. Luan, D. Qian, Accelerating approximate matrix multiplication for near-sparse matrices on gpus, Journal of Supercomputing 78 (2022) 11464–11491. URL: https://doi.org/10.1007/s11227-022-04334-5. doi:10.1007/s11227-022-04334-5.