POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

MonteCarlo Benchmarking

HIGH PERFORMANCE PROCESSORS AND SYSTEMS

TEAM MEMBERS:

Andrea Paparella - 10701904
Andrea Piras - 10725972

Academic Year: 2022-23

Contents

1 | Introduction

1.1. Overview of Monte-Carlo Simulation and Option Pricing

The Monte-Carlo option pricing simulation [3] is an algorithm for estimating financial option values. It employs random sampling and statistical analysis to generate numerous price paths for the underlying asset. Each path represents a potential future price evolution. For each path, the algorithm calculates the option's payoff at expiration using a chosen pricing model. This payoff is discounted to present value using a risk-free interest rate. By averaging the discounted payoffs across all paths, an estimate of the option's expected value is obtained. The Monte-Carlo simulation is well-suited for parallelization, particularly in multi-GPU systems. Workload distribution across multiple GPUs accelerates computation by assigning subsets of price paths to each GPU. Results are combined for the final estimated option value, enabling faster and more efficient pricing calculations.

1.2. Problem Statement

In traditional CUDA programming, developers need to explicitly manage memory transfers between the CPU and GPU. They have to allocate memory on both devices and manually copy data between them. This requires careful tracking of memory allocations, deallocations, and data transfers, which can be error-prone and time-consuming. NVIDIA's Unified Memory paradigm simplifies this process by providing a single memory space that is accessible by both the CPU and GPU, but its impact on the performance of specific workloads needs to be investigated.

1.3. Research Objectives

The primary goal of this project is to evaluate the impact of NVIDIA's Unified Memory paradigm on the performance of a specific workload, the Monte-Carlo simulation. This will be achieved by comparing the performance of a publicly available multi-GPU CUDA C++ implementation [6] that manually handles GPU device memory and data transfer with the same implementation using Unified Memory and CUDA's optimization features, such as memory prefetch and asynchronous data transfer. The project also aims to develop an equivalent Java implementation using GraalVM and the GrCUDA [4] automatic scheduler, which removes the need for end-users to deal with optimizations. This will enable us to compare the performance of different implementations.

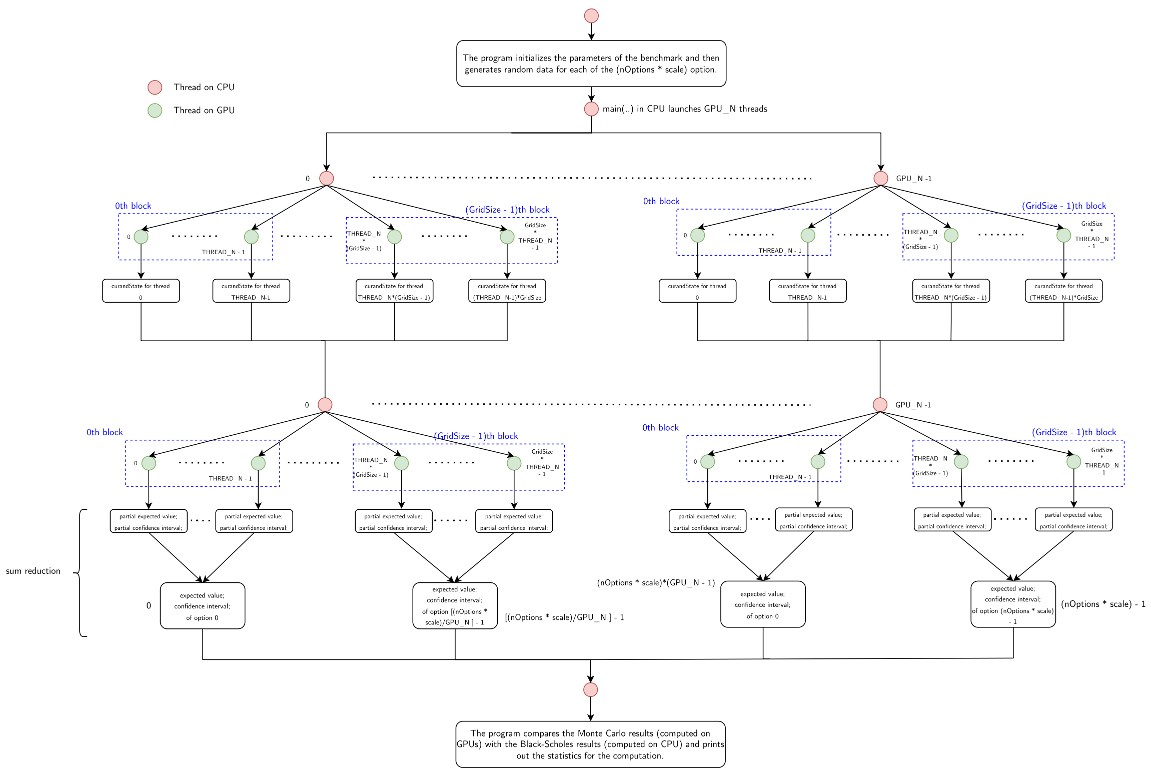
2 | The Original Multi-GPU CUDA C++ Implementation

2.1. Benchmark Program Flow

The program, after initializing the required benchmark data, can execute in two different ways: threaded or streamed.
In the threaded method, the CPU creates separate threads based on the number of GPUs. Each of these threads calls the `initMonteCarloGPU` function that initializes, through the `rngSetupStates` kernel, the `curandState` variables used for random sampling.
After that, they each call the main execution function `MonteCarloGPU`, where the `MonteCarloOneBlockPerOption` kernel is called. As the name suggests, each option computation is done on the same block. After each thread of the GPU computes its partial integral of the `callValue`, a sum reduction is performed in order to obtain the final `callValue` of the current option. After all computations have finished, the results are compared with the *Black-Scholes* formula [1] for validation.
The streamed method is similar, the main difference is that no thread is generated and the initialization of the `curandState` is called iteratively by the main thread. Also, before executing the main kernel, the program creates different `cudaStreams` based on the number of GPUs, on which the Monte-Carlo computation will be executed.
The diagrams of the two program executions are shown in figures 2.1 and 2.2.

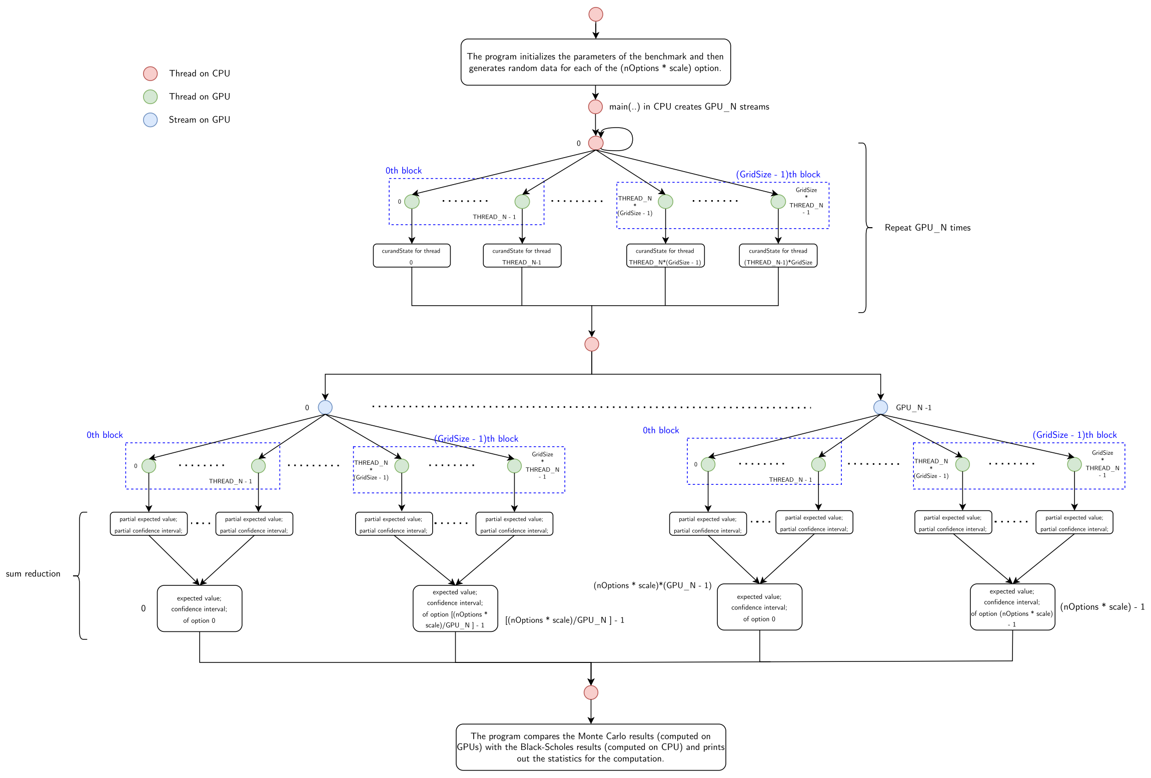
2.2. Manual handling implementation

In the original code, found in the Tartan implementation [6], memory regions are allocated manually on either the CPU or the GPUs using respectively the `cudaMallocHost` and `cudaMalloc` functions. Each memory region is bound to the device on which it was allocated.
When data needs to be migrated between devices, the `cudaMemcpyAsync` function is used. This is done before calling the `MonteCarloOneBlockPerOption` kernel since the input and output data must be migrated to the GPUs in order to read and write it. The data is therefore moved to the GPU, and after the computation is completed, it is moved back to the CPU using again the same `cudaMemcpyAsync` function.



The program initializes the parameters of the benchmark and then generates random data for each of the (nOptions * scale) option.
Thread on CPU
Thread on GPU
main(..) in CPU launches GPU_N threads
0
GPU_N -1
0th block
(GridSize - 1)th block
THREAD_N - 1
THREAD_N * (GridSize - 1)
GridSize * THREAD_N - 1
curandState for thread 0
curandState for thread THREAD_N-1
curandState for thread THREAD_N*(GridSize - 1)
curandState for thread (THREAD_N-1)*GridSize
partial expected value; partial confidence interval;
sum reduction
expected value; confidence interval; of option 0
expected value; confidence interval; of option [(nOptions * scale)/GPU_N] - 1
[(nOptions * scale)/GPU_N] - 1
(nOptions * scale)*(GPU_N - 1)
expected value; confidence interval; of option (nOptions * scale) - 1
(nOptions * scale) - 1
The program compares the Monte Carlo results (computed on GPUs) with the Black-Scholes results (computed on CPU) and prints out the statistics for the computation.

Figure 2.1: Flow of threaded execution



The program initializes the parameters of the benchmark and then generates random data for each of the (nOptions * scale) option.
Thread on CPU
Thread on GPU
Stream on GPU
main(..) in CPU creates GPU_N streams
0
0th block
(GridSize - 1)th block
THREAD_N - 1
THREAD_N * (GridSize - 1)
GridSize * THREAD_N - 1
curandState for thread 0
curandState for thread THREAD_N-1
curandState for thread THREAD_N*(GridSize - 1)
curandState for thread (THREAD_N-1)*GridSize
Repeat GPU_N times
GPU_N -1
partial expected value; partial confidence interval;
sum reduction
expected value; confidence interval; of option 0
expected value; confidence interval; of option [(nOptions * scale)/GPU_N] - 1
[(nOptions * scale)/GPU_N] - 1
(nOptions * scale)*(GPU_N - 1)
expected value; confidence interval; of option (nOptions * scale) - 1
(nOptions * scale) - 1
The program compares the Monte Carlo results (computed on GPUs) with the Black-Scholes results (computed on CPU) and prints out the statistics for the computation.

Figure 2.2: Flow of streamed execution

3 | Unified Memory (UM) Implementation

3.1. Porting to CUDA 11

In order to compile the code with CUDA 11, we modified the initial directory path within the `shared.mk` makefile from `CUDA_DIR=/usr/local/cuda-9.1/` to `CUDA_DIR=/usr/local/cuda-11.7/`. In addition, the missing libraries were installed: `lmpich`, `lmpl`. Other ones, such as `lcutil\_x86\_64` and `lnccl`, were found to be not utilized by the benchmark, so we removed them from the makefile.

3.2. Porting to Unified Memory - Version 0

In order to implement Unified Memory, memory allocation requires the use of a different method: `cudaMallocManaged`. This function allows for automatic memory allocation that is accessible to both CPUs and GPUs. This obviates the necessity for `cudaMemcpyAsync` calls, allowing them to be omitted from the code. Once the allocated memory regions are no longer required, they can be deallocated using only the `cudaFree` function. This simplifies the process as compared to the original implementation which demanded the use of `cudaFreeHost` for CPU-allocated data.

However, it's important to note that data allocation on Unified Memory does not immediately ensure accessibility for the processor using it. If the memory region resides on a different processor, a page fault can occur, resulting in the migration of data to the processor seeking access.

Our goal was to enhance the program's efficiency by diminishing memory transfer overheads that are caused by page faults in the Unified Memory setting. To accomplish this, we implemented two techniques: `cudaMemPrefetchAsync` and `cudaMemAdvise`. First, we individually tested each technique, then combined them to leverage their synergistic effects.

Details of Implementations

The complete set of code snippets for the different versions that were used throughout our analysis and experimentation can be found detailed in Chapter 8.

3.3. Optimizations

3.3.1. Prefetching the memory - Version 1

Understanding Prefetching and its Significance

The `cudaMemPrefetchAsync` directs the memory controller to initiate the transfer of a specified memory region from the CPU to the GPU, or vice versa, as early as possible. This proactive approach significantly minimizes the occurrence of page faults in the program, thereby reducing the time spent waiting for memory transfers to finish.

Prefetching Necessities and Strategy

In Figure 8.1, the `rngStates` variable is initialized by the `rngSetupStates` kernel running concurrently on the designated GPU. Proactively migrating data to the GPU before it becomes necessary helps mitigate the occurrence of page faults.

Moving on to Figure 8.4, the `MonteCarloGPU` function does the following:

1. writes to `um_OptionData` on the CPU;
2. call a kernel that reads from `um_OptionData` and writes to `um_CallValue`, on the GPU.

Point 1 suggests prefetching `um_OptionData` on the CPU. Point 2 suggests that after CPU operations on `um_OptionData` are completed, prefetching it along with `um_CallValue` on the GPU may be advantageous.

In Figure 8.8, `closeMonteCarloGPU` calculates the average and confidence interval reading `um_CallValue`, suggesting that it should be prefetched back to the CPU post-execution of the `MonteCarloOneBlockPerOption` kernel.

Challenges and Solutions in Prefetching Implementation

Throughout the execution of the `MonteCarloGPU` we observed a noticeable time span, for the initialization of `um_OptionData` on the host, during which the GPU was left substantially underutilized. Thus, we chose to anticipate the prefetching of `um_CallValue` prior to the initialization loop (Figure 8.6). This decision facilitated concurrent migration, thus anticipating the main execution kernel invocation and consequently enhancing performance.
Despite prefetching all required data, sporadic page faults still occurred during program execution. As seen on Figure 8.6, we observed that since `cudaMemPrefetchAsync` is asynchronous, the CPU may enter the initialization loop of `um_OptionData` before the migration has finished and access memory not yet available. To overcome this problem, we put a `cudaStreamSynchronize`

after the call to `cudaMemPrefetchAsync`, allowing the migration to finish before entering the loop.

3.3.2. Advising the memory migration - Version 2

Understanding Advising and Its Significance

The `cudaMemAdvise` function informs the CUDA runtime system about specific memory region usage, aiding in performance-enhancing memory management decisions. The hints include:

- `cudaMemAdviseSetReadMostly`: indicates that the memory region is likely to be read more often than written and therefore any read access by any device will create a read-only copy of the data in the processor's memory.
- `cudaMemAdviseSetPreferredLocation`: indicates the preferred location for the memory region, guiding the data migration when a page fault occurs.
- `cudaMemAdviseSetAccessedBy`: indicates that the memory region will be accessed by a specific device and causes data to be always mapped on specified processor's page tables. No data is migrated in this case.

Advising Necessities and Strategy

We followed a similar reasoning as the one applied in version 1: `um_OptionData` should reside on CPU memory, accessed via GPU mappings on the page table during kernel execution, avoiding migration except for the initial one to the host.

Similar logic applies to `um_CallValue`, suggesting residence on GPU memory initially, moving to the CPU after the main kernel execution.

Challenges and Solutions in Advising Implementation

The initial placement of `cudaMemAdvise` on `um_CallValue` within `MonteCarloGPU` inadvertently prompted page faults due to its asynchronous execution before the main kernel finished; when a page fault occurred, the memory was then migrated to the CPU instead of the GPU. Relocating `cudaMemAdvise` to `closeMonteCarloGPU` post `cudaDeviceSynchronize` invocation allowed the kernel to complete prior to the `cudaMemAdvise` call, avoiding the issue.

Considering `um_CallValue`, we examined two alternatives:

- Utilizing `cudaMemAdviseSetPreferredLocation` hint with `cudaCpuDeviceId`, and `cudaMemAdviseSetAccessedBy` with the device ID.

- Using the same previous hints, but assigning GPU as the preferred location, suggesting CPU to map `um_CallValue` on its page table.

Option one increased the number of GPU page faults, resulting in 25% more migration time as seen in Figure 6.5. Option two resulted in performance degradation on almost all metrics, due to CUDA's guidance [8] on potential host memory migration upon CPU data access. Both approaches were thus disregarded.

Attempts to apply `cudaMemAdviseSetReadMostly` hint throughout various code points resulted in either performance decline or stasis, as no memory region was read more than once by multiple devices or both device and host. Thus, this hint was omitted from the final implementation.

3.3.3. Combining Prefetching and Advising - Version 3

Understanding Synergies

Though `cudaMemPrefetchAsync` effectively prevents page faults, the overhead of calling it and memory migrations can be significant. We attempted to amalgamate the strategies from the previous versions to diminish the usage of `cudaMemPrefetchAsync`, leveraging `cudaMemAdvise` to prevent page faults caused by the removal of the aforementioned function calls.

Synergies Necessities and Strategy

The only successful prefetch call elimination without performance degradation was for `um_OptionData` post-CPU initialization. Replacing this call with the version 2 strategy (`cudaMemAdvise` using the `cudaMemAdviseSetAccessedBy` hint) eliminated the need for migration from the CPU to the GPU thanks to the remote mappings and thus prevented page faults, which can be observed in Figure 6.4.

Challenges and Solutions in Synergies Implementation

Profiling showed that remaining `cudaMemPrefetchAsync` calls for `um_OptionData` were disproportionately time-consuming (Figure 6.8). Substituting them with version 2's `cudaMemAdvise` decreased prefetch time but increased CPU page faults. However, due to the longer duration of page fault-induced migrations, we discarded this function replacement. Applying this logic to `rngStates` and `um_CallValue` was not feasible due to their shorter prefetch times and the lengthy duration of page fault-induced migrations.

4 | GrCUDA: Java Implementation

4.1. Exploring the Potential of GrCUDA

GrCUDA [4] is a polyglot GPU execution framework that provides a way to execute CUDA kernels from languages running on GraalVM, such as Java, Python, and others. It manages memory transfers between the CPU and GPU transparently, thereby allowing users to concentrate on writing the CUDA kernels without worrying about the details of data transfer. This effectively democratizes the development of multi-GPU applications.

4.2. Porting to Java

4.2.1. Program Structure

We adopted *Politecnico di Milano*'s *NectsLab* implementation [7] for scalable, modular benchmark management. We translated the code from C++ to Java, adhering to the *Tartan* benchmark's style and structure, enabling reliable performance comparison. This helped in isolating any performance differences between the two languages. If the Java version ran slower or faster, we could confidently attribute these differences to the languages themselves or their respective runtime environments, rather than structural differences in the code.
We used Maven for a streamlined build process and consistent library/dependency usage.

4.2.2. Program Design

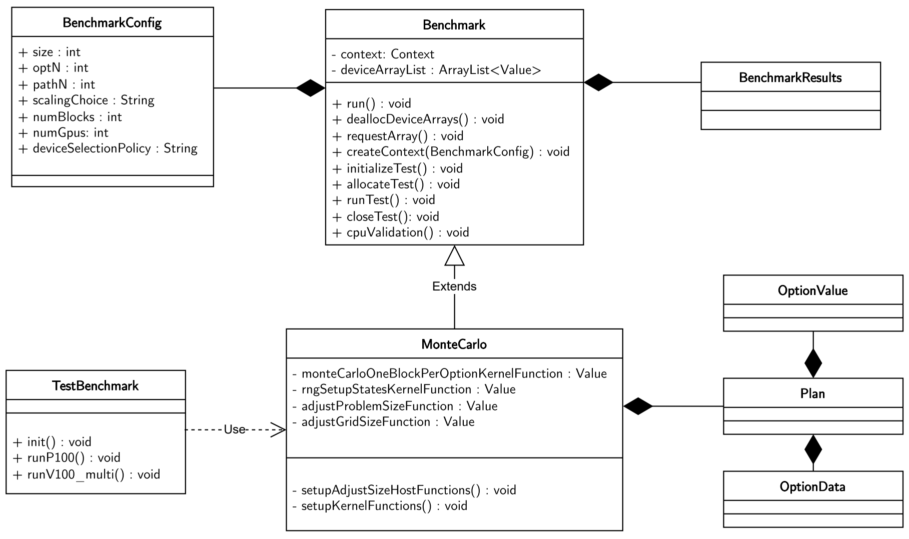

BenchmarkConfig
+ size : int
+ optN : int
+ pathN : int
+ scalingChoice : String
+ numBlocks : int
+ numGpus: int
+ deviceSelectionPolicy : String
Benchmark
- context: Context
- deviceArrayList : ArrayList<Value>
+ run() : void
+ deallocDeviceArrays() : void
+ requestArray() : void
+ createContext(BenchmarkConfig) : void
+ initializeTest() : void
+ allocateTest() : void
+ runTest() : void
+ closeTest(): void
+ cpuValidation() : void
BenchmarkResults
Extends
OptionValue
MonteCarlo
- monteCarloOneBlockPerOptionKernelFunction : Value
- rngSetupStatesKernelFunction : Value
- adjustProblemSizeFunction : Value
- adjustGridSizeFunction : Value
- setupAdjustSizeHostFunctions() : void
- setupKernelFunctions() : void
TestBenchmark
+ init() : void
+ runP100() : void
+ runV100_multi() : void
Use
Plan
OptionData

Figure 4.1: UML of the GrCUDA MonteCarlo program

The **MonteCarlo** class, extending the **Benchmark** class, provides functions and attributes for benchmark management, such as:

- GraalVM `context`: generated according to the given configuration.
- `deviceArrayList`: comprises GrCUDA arrays.
- `run()`: controls the test flow, coordinating operations like allocation, initialization, execution, and closure. Results are logged to a CSV file post each iteration. After all iterations, device arrays are deallocated, and the context is closed.
- `createContext()`: arranges a GraalVM context with GrCUDA-specific options from `BenchmarkConfig`, including various policies and device selection.

The **MonteCarlo** class overrides inherited methods:

- `init()`: prepares the benchmark by setting host functions and kernel functions. It calculates total options for the run, initiates an array of plans, and assigns option ranges, callValue ranges, and other parameters to each plan.
- `setupAdjustSizeHostFunctions()` and `setupKernelFunctions()`: bind host and kernel functions from external files, using GraalVM's Polyglot API and Value interface.
- `allocateTest(int iteration)`: allocates memory for benchmark arrays, iterating over each GPU according to configuration parameters.
- `initializeTest(int iteration)`: initiates CUDA `rngStates` as defined in the configuration, by executing the kernel function `rngSetupStatesKernelFunction`.
- `runTest(int iteration)`: carries out the Monte-Carlo simulation. It goes through each GPU and option, calculating parameters and executing the kernel function `monteCarloOneBlockPerOptionKernelFunction`.
- `closeTest(int iteration)`: retrieves results post-simulation, computes averages and confidence intervals, and stores them in the `Plan` instance.
- `cpuValidation()`: validates GPU results against CPU outcomes using *Black-Scholes* computation. It computes an L1 norm to quantify any difference and throws an exception if the difference exceeds a set threshold.

4.3. Best Practice: Binding from PTX files

GrCUDA supports dynamic kernel creation from a Java string of CUDA C++ code, as showcased in *NectsLab*'s GrCUDA repository.

Here is an example in C++ for context initialization and building the kernels:

```
Value buildKernel = context.eval("grcuda", "buildkernel");
Value kernelFuntion = buildKernel.execute("<kernel_code_written_in_C++>", "kernelName", "param1_type, paramN_type");
```

However, our Monte Carlo kernels make use of complex data structures and external library functions, and their import is not supported by the binding from the Java string.

Specifically, our kernels leverage `curandState`, `curand_init()`, and `curand_normal()` functions from `helper_cuda.h` and `curand_kernel.h` libraries. To overcome the import constraint, we bounded functions from PTX files. Our CUDA kernels, including functions from `helper_cuda.h` and `curand_kernel.h`, were written separately, compiled into a PTX file, and linked using a binding function. The compilation command was:

```
nvcc -ptx -gencode=arch=compute_XX,code=sm_XX -o output.ptx input.cu -I../../common/inc/
```

`XX` should be replaced based on the GPU used. The `input.cu` file contains the C++ code to compile. After compilation, the resulting `output.ptx` is used for binding.

To invoke the kernels from the host language, the following code was used:

```
Value cu = context.eval("grcuda", "CU");
Value kernelFunction = cu.invokeMember("bindkernel", path_to_binded_kernels_PTX_file,
        "cxx kernelFunctionName(<param1>: <type>, <paramN>: <type>): <return_type>");
```

Additionally, specific host functions employ `cuda_runtime` and `helper_cuda.h` library functions, `setupAdjustSizeHostFunctions()` and `setupKernelFunctions()`. To accommodate this, we placed these functions in a separate file, compiled into a Shared Object file using:

```
nvcc -o output.so -std=c++11 --shared -Xcompiler -fPIC input.cpp -I../../common/inc/
```

In this case, the `invokeMember` function was modified as follows:

```
Value hostFunction = cu.invokeMember("bind", path_to_binded_host_functions_SO_file,
            "cxx hostFucntionName(<param1>: <type>, <paramN>: <type>): <return_type>");
```

4.4. Optimizations

In Chapter 6, we highlight the inferior performance of GrCUDA in contrast to the Unified Memory version, despite various configuration alterations.

Our analysis then shifted to the benchmark's memory data transfer. Noticing a significant number of page faults during data copying from GPU to CPU (see Figure 6.10), we attempted prefetching within the code using explicit calls, in order to mitigate this issue. However, prefetching failed to enhance performance significantly, leading to its removal. In the end, we allowed GrCUDA's environment to manage its optimization, as it more closely aligns with the goal of simplifying the integration of CUDA into other languages.

5 | Methodology

5.1. Performance Evaluation Metrics

The evaluation of the Manual, Unified Memory, and GrCuda implementations focused on three factors: average initialization time, average execution time, and page faults.
The initialization time was measured as the time required for memory allocation and initialization of the `curandStates` in the `rngSetupStates` kernel, i.e. the execution of `initMonteCarloGPU` function.
The execution time was instead measured as the time taken to compute the main algorithmic computations in the `MonteCarloOneBlockPerOption` kernel and to store the results back to the CPU memory, excluding the program CPU validation, i.e. the execution of `MonteCarloGPU` and `closeMonteCarloGPU` functions.
Other factors considered, particularly for the implementation of Unified Memory with `cudaMemAdvise`, included the total time required for `DeviceToHost` and `HostToDevice` memory migrations and the time required to complete the CUDA API calls.

5.2. Data Collection and Analysis Tools

The initialization and execution times were measured using `steady_clock` from the `chrono` library. The number of page faults was determined using *nvprof* [10], a command-line profiler for CUDA applications provided by NVIDIA. *Nsight Systems* [9], a graphic performance analysis tool, was also utilized to visualize CPU-GPU interactions, track GPU activity, and trace GPU workloads.
These tools facilitated the identification of areas for optimization and fine-tuning, in order to improve the performance of the algorithm.

5.3. Testing Procedure

Due to the increasing input sizes, it was observed that the overall program execution time significantly increased compared to smaller sizes. Consequently, to maintain feasibility and optimize efficiency, the average times were calculated averaging 5 executions, as opposed to the 10 executions used for smaller sizes.
Also, in order to account for the cold start effect of the GPUs, an additional run was conducted which was not included in the reported results.

6 | Results and Discussion

Note: For brevity, we use "initialization time" to refer to the average initialization time, and "execution time" to refer to the average execution time.

6.1. Preliminary Considerations

Through the porting to Unified Memory and the implementation of prefetching and advising strategies, we achieved a significant reduction in initialization time. However, the execution time remained roughly the same for all versions tested.
This is because the kernel execution time obscured the speedup achieved by eliminating CPU and GPU page faults. This phenomenon became more pronounced as the input size increased, as seen in Figure 6.11. Nevertheless, we maintained our focus on minimizing the number of page faults, despite it not substantially contributing to the reduction of execution time.

6.2. Performance Comparison: Baseline vs. UM

The marked improvement in initialization times was primarily due to the `cudaMallocHost` and `cudaMalloc` functions, which required considerably more time than the `cudaMallocManaged` function.

This difference is also evident when comparing times across various numbers of GPUs between versions. In any UM version, as illustrated in Figure 6.14, the initialization time with 4 GPUs was almost twice that with 1 GPU. In contrast, this difference was less pronounced in the baseline version, as shown in Figure 6.12. This can be attributed to the `rngSetupStates` kernel execution time being an order of magnitude smaller than the `cudaMallocHost` execution time. As a result, running 1 kernel instead of 4 did not significantly influence the overall initialization time, as the `cudaMallocHost` function acted as a bottleneck.

This behavior was also observed when considering size variation with a fixed number of GPUs, as detailed in Subsection 6.9.2. Unlike non-UM behavior, on Pascal and later architectures managed memory may not be physically allocated when `cudaMallocManaged` returns; it may only be populated on access. In other words, pages and page table entries may not be created until they are accessed by the GPU or the CPU [2]. This factor led to an increase in the time required by the `cudaMallocHost` and `cudaMalloc` functions as size escalated. However, the `cudaMallocManaged` function continued to be relatively scalable and efficient since it avoided allocating memory regions that are not accessed in the `initMonteCarloGPU` function.

6.3. Performance Comparison: Version 0 vs. 1

Prefetching improved initialization time by almost 6ms as evidenced by the difference between *Nsight* reports in Figure 6.6 and 6.7. Moreover, substantial reductions in page faults were observed compared to version 0, as reported in Figure 6.1 and 6.2.
Notice that this new strategy showed new patterns of data transfer between host and device. In version 0, there were small transfers (average sizes around 120-170KB), while in version 1, there were fewer, larger transfers (2MB each).

6.4. Performance Comparison: Version 0 vs. 2

Having set the preferred location of `rngStates` on the device guided the migration policy upon encountering a fault within that memory region. This led to a slight reduction in the initialization time compared to version 0, as evidenced by Figure 6.20. Further, by setting the host as the preferred location and granting GPU access to `um_OptionData`, we eliminated GPU faults and created remote mappings. Coupled with other memory advisories, this strategy resulted in a halving of CPU faults, which is proved by the *nvprof* analysis depicted in Figure 6.3.

6.5. Performance Comparison: Version 0 vs. 3

The combination of prefetching and advising led to the best timings for the initialization (see Figure 6.20).
Furthermore, GPU page faults of version 0 were avoided in version 3, thanks to the utilization of "Remote mapping from device", as seen in Figure 6.4. Notably, this version displayed data transfer patterns between host and device akin to those found in version 1. This similarity can be attributed to the retention of the majority of improvements from version 1, combined with the minimal integration of enhancements from version 2.

6.6. Performance Comparison: Version 0 vs. GrCUDA

Transitioning to GrCUDA resulted in a performance deterioration with respect to C++ code with Unified Memory in comparative tests. This gap was due to GrCUDA's limited capacity for granular CUDA code modifications. Despite GrCUDA's aim to simplify heterogeneous computing with a unified runtime, it may not offer the same level of control and optimization afforded by lower-level languages like C++.
While there was a performance drop, the decrease became less pronounced as the size increased.

This trend can be observed in initialization and execution time, as illustrated in Figures 6.21 and 6.20. In addition, for sizes larger than 1024, the initialization time was shorter compared to the baseline.

6.7. Validation of Data Transfer Patterns

The *nvprof* outputs in Section 6.8.1 indicate that each version - that uses prefetching - involved a total data transfer size of 8 MB from the host to the device and 4 MB from the device to the host. We have confirmed the validity of these figures using a multi-step verification process:

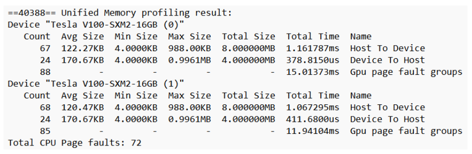
- We began by scrutinizing the code and carrying out calculations to account for the quantity of prefetched memory. This involved considering the number of bytes each struct takes in each prefetch function.
- We further extended our analysis by examining the *Nsight* output.
- We then focused on adding up only the actual data that was moved.
- Finally, we compared our manually calculated figures with the outputs from *nvprof*.

Our analysis confirmed the *nvprof* outputs' accuracy and affirmed that the data transfer mechanism between the host and the device aligned with the specifications in the official documentation [5]. Specifically, when a memory region is prefetched for the first time and no physical memory has been previously allocated for this region, the memory is directly populated and mapped on the destination device. This implies that the data was not moved from host to device (HtoD) or from device to host (DtoH). This mechanism was beneficial as it resulted in notable time savings.
Similar reasonings could be also applied to other versions when page faults occurred, considering the behavior of `cudaMallocManaged` described in section 6.2.

6.8. Profilings

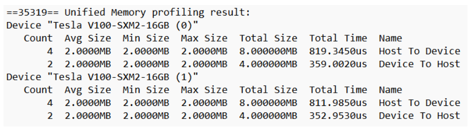
6.8.1. Nvprof Outputs

The following outputs have been generated executing the benchmark, executed with a 2 GPU, Streamed Strong, 1024 configuration. The same patterns are noted in Streamed Weak and Threaded modalities and are not shown for brevity.



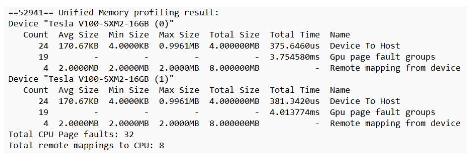
```
==40388== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      67  122.27KB  4.0000KB  988.00KB  8.000000MB  1.161787ms  Host To Device
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  378.8150us  Device To Host
      88         -         -         -           -  15.01373ms  Gpu page fault groups
Device "Tesla V100-SXM2-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      68  120.47KB  4.0000KB  988.00KB  8.000000MB  1.067295ms  Host To Device
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  411.6800us  Device To Host
      85         -         -         -           -  11.94104ms  Gpu page fault groups
Total CPU Page faults: 72
```

Figure 6.1: Version 0



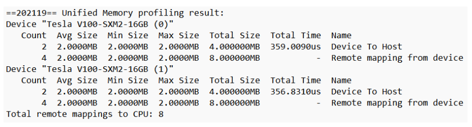
```
==35319== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB  819.3450us  Host To Device
       2  2.0000MB  2.0000MB  2.0000MB  4.000000MB  359.0020us  Device To Host
Device "Tesla V100-SXM2-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB  811.9850us  Host To Device
       2  2.0000MB  2.0000MB  2.0000MB  4.000000MB  352.9530us  Device To Host
```

Figure 6.2: Version 1



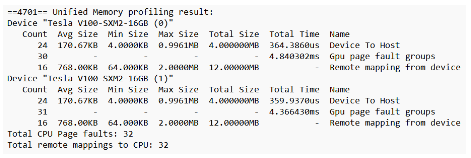
```
==52941== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  375.6460us  Device To Host
      19         -         -         -           -  3.754580ms  Gpu page fault groups
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB           -  Remote mapping from device
Device "Tesla V100-SXM2-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  381.3420us  Device To Host
      19         -         -         -           -  4.013774ms  Gpu page fault groups
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB           -  Remote mapping from device
Total CPU Page faults: 32
Total remote mappings to CPU: 8
```

Figure 6.3: Version 2



```
==202119== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  2.0000MB  2.0000MB  2.0000MB  4.000000MB  359.0090us  Device To Host
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB           -  Remote mapping from device
Device "Tesla V100-SXM2-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  2.0000MB  2.0000MB  2.0000MB  4.000000MB  356.8310us  Device To Host
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB           -  Remote mapping from device
Total remote mappings to CPU: 8
```

Figure 6.4: Version 3



```
==4701== Unified Memory profiling result:
Device "Tesla V100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  364.3860us  Device To Host
      30         -         -         -           -  4.840302ms  Gpu page fault groups
      16  768.00KB  64.000KB  2.0000MB  12.00000MB           -  Remote mapping from device
Device "Tesla V100-SXM2-16GB (1)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  359.9370us  Device To Host
      31         -         -         -           -  4.366430ms  Gpu page fault groups
      16  768.00KB  64.000KB  2.0000MB  12.00000MB           -  Remote mapping from device
Total CPU Page faults: 32
Total remote mappings to CPU: 32
```

Figure 6.5: Version 2 - Option One

6.8.2. Nsight System Outputs

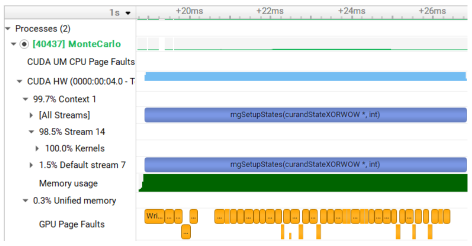


Figure 6.6: Init, Strong Streamed, 2 GPUs, 1024, Version 0

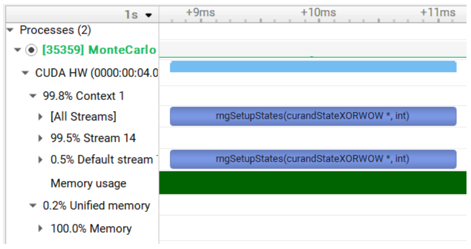


Figure 6.7: Init, Strong Streamed, 2 GPUs, 1024, Version 1

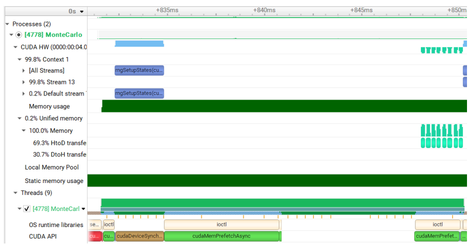


Figure 6.8: Memory transfers HtoD, Strong Streamed, 1 GPU, 1024, Version 1

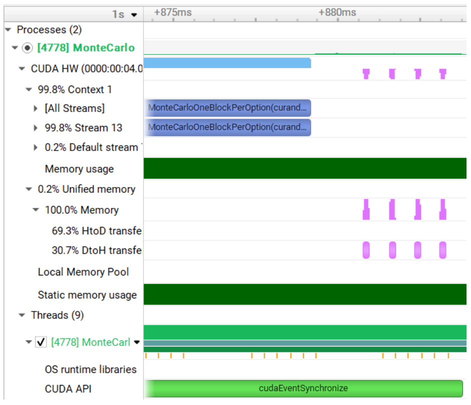


Figure 6.9: Memory transfers DtoH, Strong Streamed, 1 GPU, 1024, Version 1

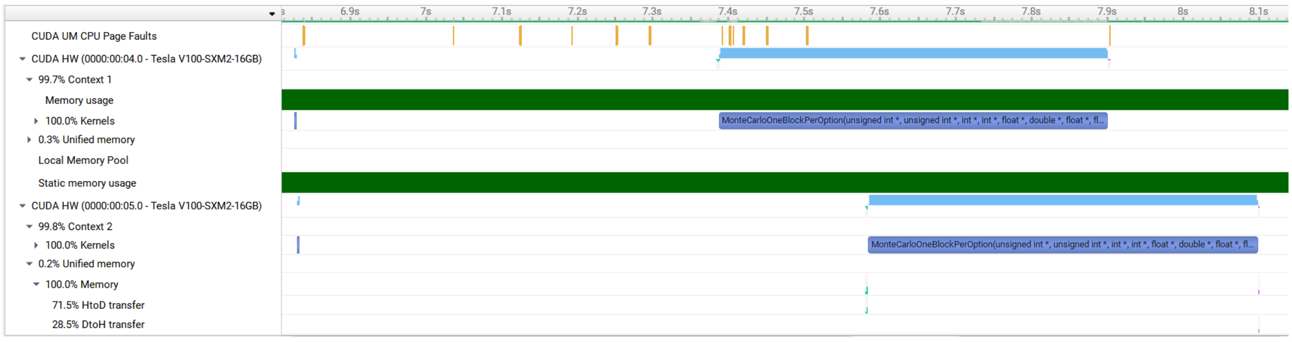

CUDA UM CPU Page Faults
CUDA HW (0000:00:04.0 - Tesla V100-SXM2-16GB)
99.7% Context 1
Memory usage
100.0% Kernels
0.3% Unified memory
Local Memory Pool
Static memory usage
CUDA HW (0000:00:05.0 - Tesla V100-SXM2-16GB)
99.8% Context 2
100.0% Kernels
0.2% Unified memory
100.0% Memory
71.5% HtoD transfer
28.5% DtoH transfer
6.9s 7s 7.1s 7.2s 7.3s 7.4s 7.5s 7.6s 7.7s 7.8s 7.9s 8s 8.1s
MonteCarloOneBlockPerOption(unsigned int *, unsigned int *, int *, int *, float *, double *, float *, fl...
MonteCarloOneBlockPerOption(unsigned int *, unsigned int *, int *, int *, float *, double *, float *, fl...

Figure 6.10: Overall, Strong, 2 GPUs, 1024, GrCUDA version

6.9. Timings Graphs

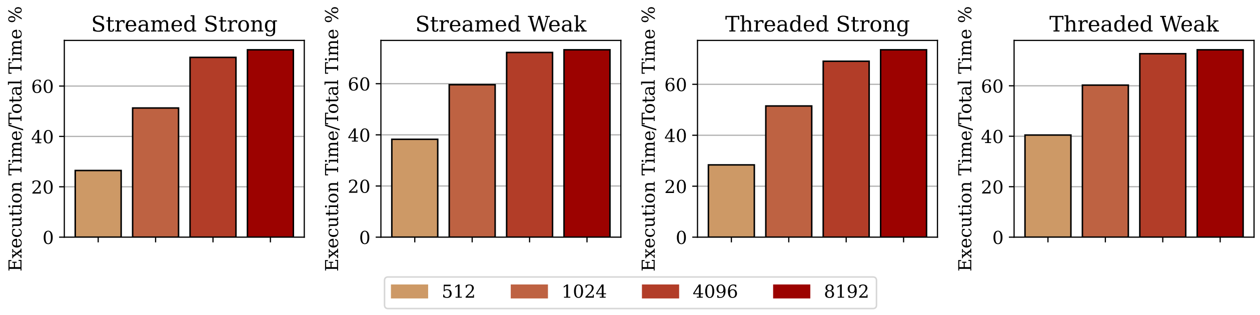


Figure 6.11: Baseline, Execution Time Ratio

6.9.1. GPU-varied Graphs

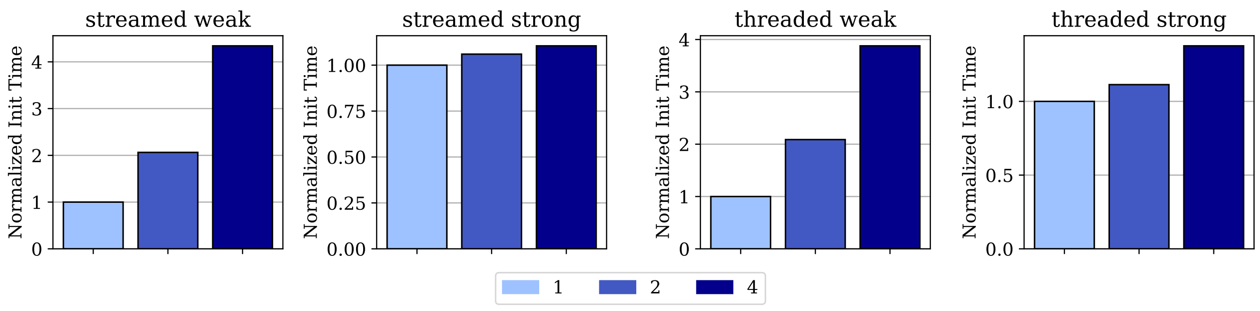


Figure 6.12: Baseline, 8192

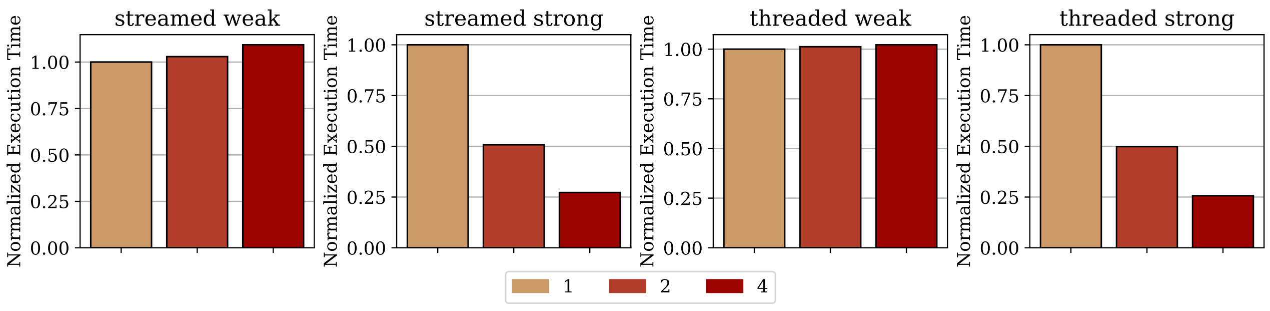


Figure 6.13: Baseline, 8192

Note that other UM versions exhibit a similar trend and are not shown for brevity.

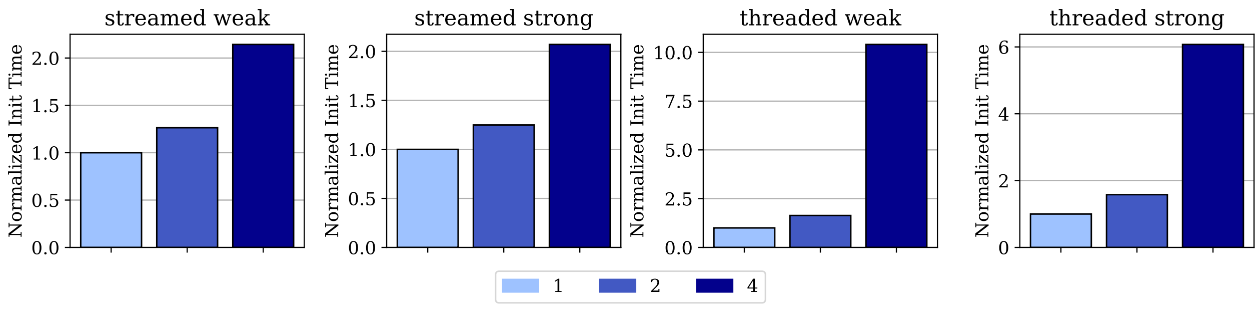


Figure 6.14: UMv3, 8192

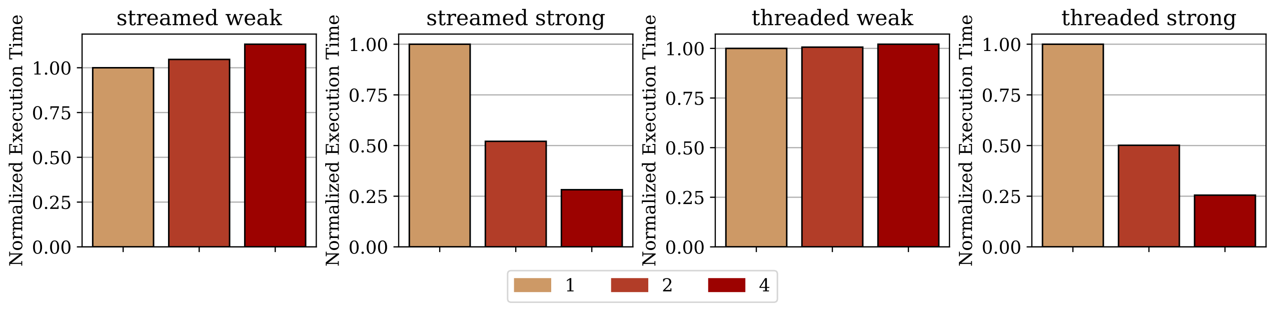


Figure 6.15: UMv3, 8192

6.9.2. Size-varied Graphs

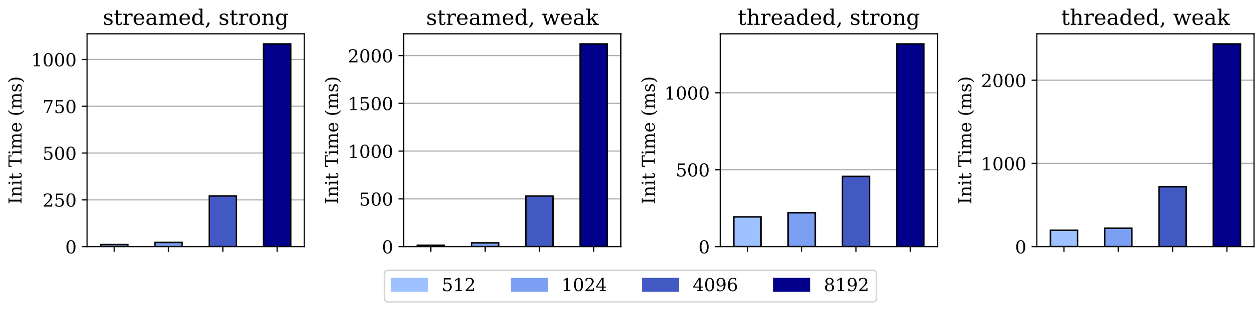


Figure 6.16: Baseline, 2 GPUs

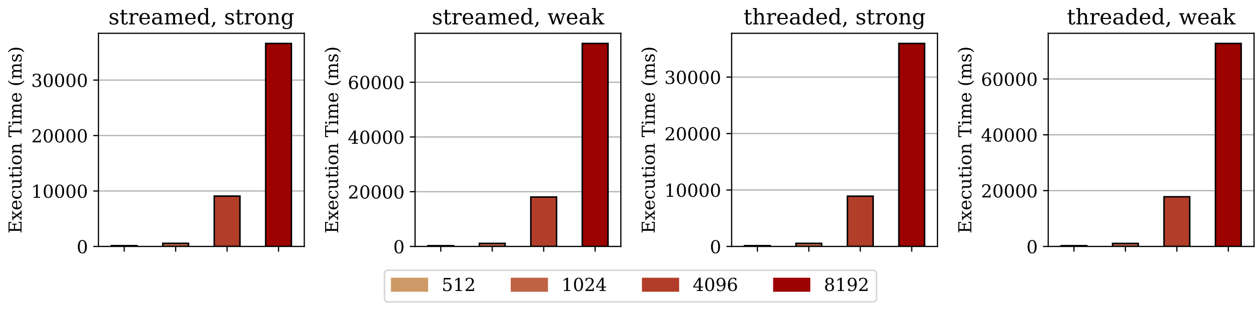


Figure 6.17: Baseline, 2 GPUs

Note that other UM versions exhibit a similar trend and are not shown for brevity.

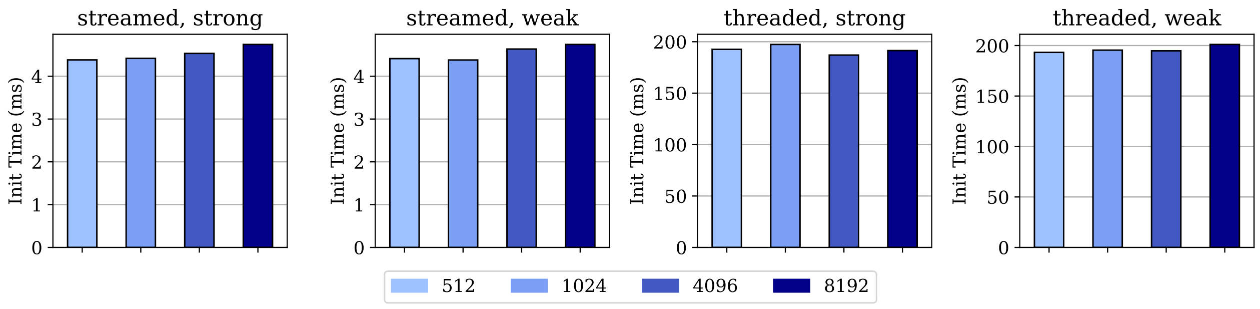


Figure 6.18: UMv3, 2 GPUs

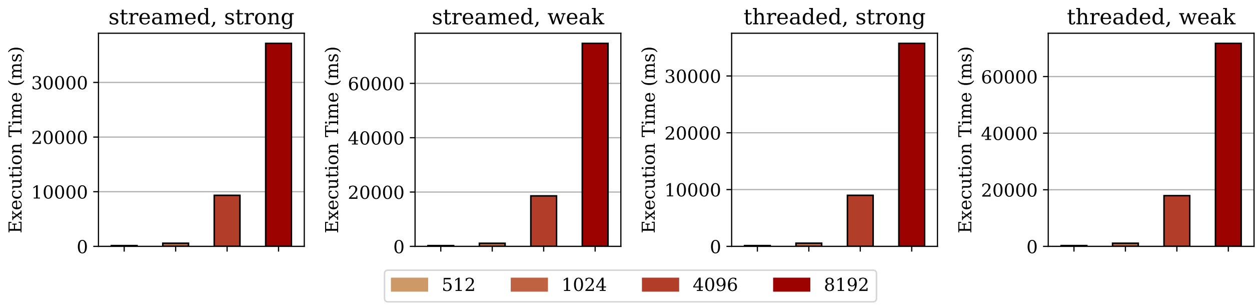


Figure 6.19: UMv3, 2 GPUs

6.9.3. Version-varied Graphs

Note that other configurations exhibit a similar trend and are not shown for brevity.

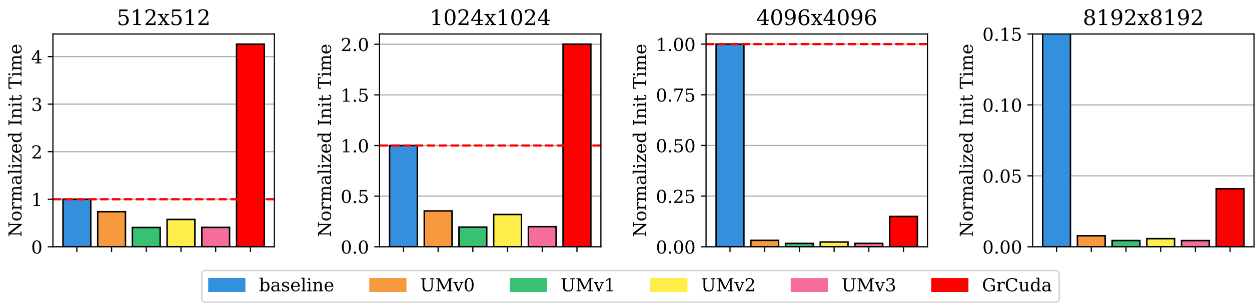


Figure 6.20: Streamed Strong, 2 GPUs

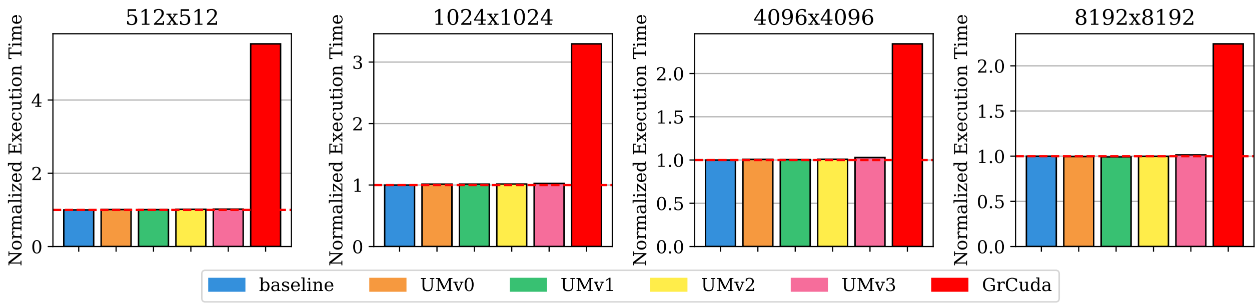


Figure 6.21: Streamed Strong, 2 GPUs

7 | Conclusion

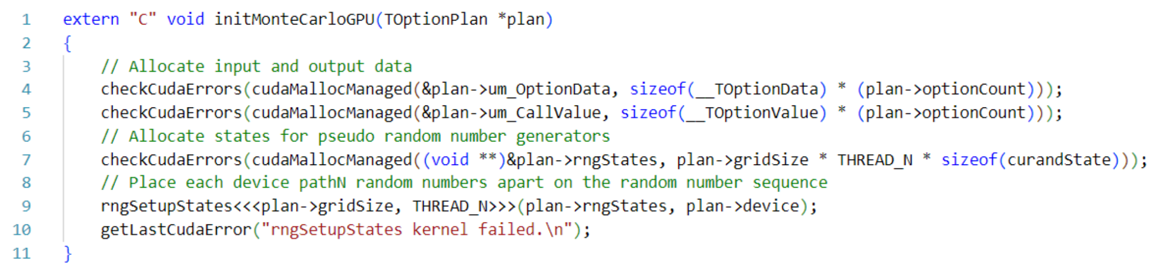
This report explored the process of transitioning to Unified Memory (UM) from a standard CUDA programming model. We conducted a comprehensive analysis of different versions, from the baseline to a fully optimized state, leveraging various techniques including memory prefetching, advising strategies, and also employing a framework like GrCUDA.

Throughout the versions tested, the use of Unified Memory and optimization techniques like prefetching and memory advising significantly improved initialization times and minimized page faults. This optimization was even more evident when variables like the number of GPUs or size were introduced. However, we noted a relative steadiness in the execution time across the versions, indicating that mitigating page faults does not necessarily correlate with performance improvements, particularly for kernels with extensive runtimes in comparison to data transfer time.

When shifting to GrCUDA, we faced a performance setback due to its limited granularity in relation to CUDA code modifications. Yet, it demonstrated a potential for scalability, as the decrease in performance became less pronounced with larger input sizes.

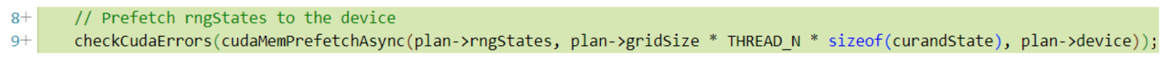
In conclusion, the transition to Unified Memory presents a viable approach to improving performance in CUDA applications. Furthermore, using higher-level frameworks like GrCUDA, despite potential performance trade-offs, could offer simplified programming for heterogeneous computing.

8 | Code Listings



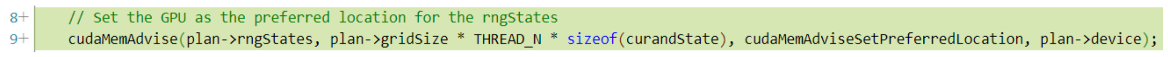
```
extern "C" void initMonteCarloGPU(TOptionPlan *plan)
{
    // Allocate input and output data
    checkCudaErrors(cudaMallocManaged(&plan->um_OptionData, sizeof(__TOptionData) * (plan->optionCount)));
    checkCudaErrors(cudaMallocManaged(&plan->um_CallValue, sizeof(__TOptionValue) * (plan->optionCount)));
    // Allocate states for pseudo random number generators
    checkCudaErrors(cudaMallocManaged((void **)&plan->rngStates, plan->gridSize * THREAD_N * sizeof(curandState)));
    // Place each device pathN random numbers apart on the random number sequence
    rngSetupStates<<<plan->gridSize, THREAD_N>>>(plan->rngStates, plan->device);
    getLastCudaError("rngSetupStates kernel failed.\n");
}
```

Figure 8.1: Baseline version, Initialization function



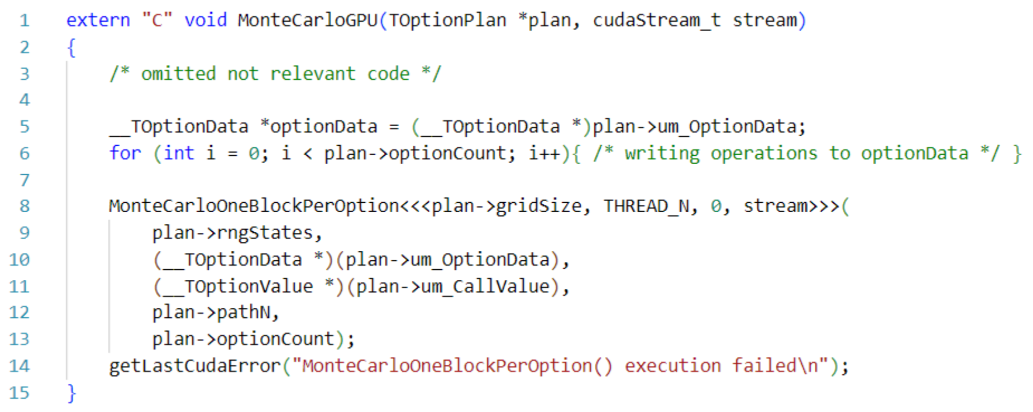
```
    // Prefetch rngStates to the device
    checkCudaErrors(cudaMemPrefetchAsync(plan->rngStates, plan->gridSize * THREAD_N * sizeof(curandState), plan->device));
```

Figure 8.2: Prefetching code lines added to initialization function



```
    // Set the GPU as the preferred location for the rngStates
    cudaMemAdvise(plan->rngStates, plan->gridSize * THREAD_N * sizeof(curandState), cudaMemAdviseSetPreferredLocation, plan->device);
```

Figure 8.3: Advising code lines added to initialization function

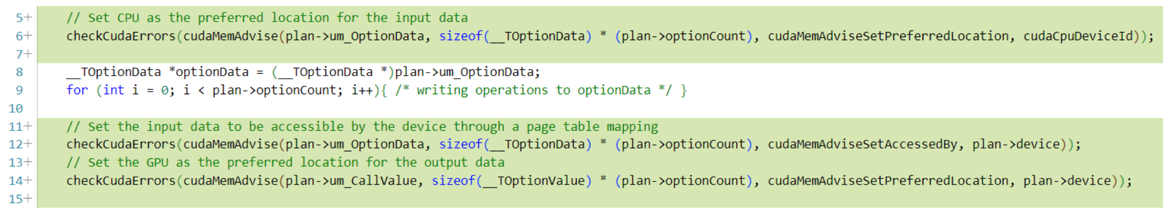


```
extern "C" void MonteCarloGPU(TOptionPlan *plan, cudaStream_t stream)
{
    /* omitted not relevant code */

    __TOptionData *optionData = (__TOptionData *)plan->um_OptionData;
    for (int i = 0; i < plan->optionCount; i++){ /* writing operations to optionData */ }

    MonteCarloOneBlockPerOption<<<plan->gridSize, THREAD_N, 0, stream>>>(
        plan->rngStates,
        (__TOptionData *)(plan->um_OptionData),
        (__TOptionValue *)(plan->um_CallValue),
        plan->pathN,
        plan->optionCount);
    getLastCudaError("MonteCarloOneBlockPerOption() execution failed\n");
}
```

Figure 8.4: Baseline version, Execution function



```
    // Set CPU as the preferred location for the input data
    checkCudaErrors(cudaMemAdvise(plan->um_OptionData, sizeof(__TOptionData) * (plan->optionCount), cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId));

    __TOptionData *optionData = (__TOptionData *)plan->um_OptionData;
    for (int i = 0; i < plan->optionCount; i++){ /* writing operations to optionData */ }

    // Set the input data to be accessible by the device through a page table mapping
    checkCudaErrors(cudaMemAdvise(plan->um_OptionData, sizeof(__TOptionData) * (plan->optionCount), cudaMemAdviseSetAccessedBy, plan->device));
    // Set the GPU as the preferred location for the output data
    checkCudaErrors(cudaMemAdvise(plan->um_CallValue, sizeof(__TOptionValue) * (plan->optionCount), cudaMemAdviseSetPreferredLocation, plan->device));

```

Figure 8.5: Advising code lines added to execution function

```
extern "C" void MonteCarloGPU(TOptionPlan *plan, cudaStream_t stream)
{
    /* omitted not relevant code */

    // Prefetch the input data to the CPU
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionData *)(plan->um_OptionData), plan->optionCount * sizeof(__TOptionData), cudaCpuDeviceId, stream));
    // Wait for prefetch to finish
    checkCudaErrors(cudaStreamSynchronize(stream));
    // Prefetch the output data to the device
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionValue *)(plan->um_CallValue), plan->optionCount * sizeof(__TOptionValue), plan->device, stream));

    __TOptionData *optionData = (__TOptionData *)plan->um_OptionData;
    for (int i = 0; i < plan->optionCount; i++){ /* writing operations to optionData */ }

    // Prefetch the input data to the device
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionData *)(plan->um_OptionData), plan->optionCount * sizeof(__TOptionData), plan->device, stream));

    MonteCarloOneBlockPerOption<<<plan->gridSize, THREAD_N, 0, stream>>>(
        plan->rngStates,
        (__TOptionData *)(plan->um_OptionData),
        (__TOptionValue *)(plan->um_CallValue),
        plan->pathN,
        plan->optionCount);
    getLastCudaError("MonteCarloOneBlockPerOption() execution failed\n");

    // Prefetch the output data to the CPU
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionValue *)(plan->um_CallValue), plan->optionCount * sizeof(__TOptionValue), cudaCpuDeviceId));
}
```

Figure 8.6: Prefetching code lines added to execution function

```
extern "C" void MonteCarloGPU(TOptionPlan *plan, cudaStream_t stream)
{
    /* omitted not relevant code */

    // Prefetch the input data to the CPU
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionData *)(plan->um_OptionData), sizeof(__TOptionData) * (plan->optionCount), cudaCpuDeviceId, stream));
    // Wait for the prefetch to finish
    checkCudaErrors(cudaStreamSynchronize(stream));
    // Prefetch the output data to the device
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionValue *)(plan->um_CallValue), plan->optionCount * sizeof(__TOptionValue), plan->device, stream));

    __TOptionData *optionData = (__TOptionData *)plan->um_OptionData;
    for (int i = 0; i < plan->optionCount; i++){ /* writing operations to optionData */ }

    // Set the input data to be accessible by the device through a page table mapping
    checkCudaErrors(cudaMemAdvise(plan->um_OptionData, sizeof(__TOptionData) * (plan->optionCount), cudaMemAdviseSetAccessedBy, plan->device));

    MonteCarloOneBlockPerOption<<<plan->gridSize, THREAD_N, 0, stream>>>(
        plan->rngStates,
        (__TOptionData *)(plan->um_OptionData),
        (__TOptionValue *)(plan->um_CallValue),
        plan->pathN,
        plan->optionCount);
    getLastCudaError("MonteCarloOneBlockPerOption() execution failed\n");

    // Prefetch the output data to the CPU
    checkCudaErrors(cudaMemPrefetchAsync((__TOptionValue *)(plan->um_CallValue), plan->optionCount * sizeof(__TOptionValue), cudaCpuDeviceId, stream));
}
```

Figure 8.7: Prefetching and Advising code lines added to execution function

```
extern "C" void closeMonteCarloGPU(TOptionPlan *plan)
{
    for (int i = 0; i < plan->optionCount; i++)
    { /* read operations to plan->optionData and plan->um_CallValue + writing operations plan->callValue */ }

    checkCudaErrors(cudaFree(plan->rngStates));
    checkCudaErrors(cudaFree(plan->um_CallValue));
    checkCudaErrors(cudaFree(plan->um_OptionData));
}
```

Figure 8.8: Baseline version, Close function

```
    // Set the CPU as the preferred location for the output data
    checkCudaErrors(cudaMemAdvise(plan->um_CallValue, sizeof(__TOptionValue) * (plan->optionCount), cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId));

```

Figure 8.9: Advising code lines added to close function

References

[1] Black-scholes model. URL https://en.wikipedia.org/wiki/Black-Scholes_model.

[2] Unified memory for cuda beginners. URL https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[3] Monte-carlo for option pricing. URL https://en.wikipedia.org/wiki/Monte_Carlo_methods_for_option_pricing.

[4] Grcuda - a polyglot language binding for cuda in graalvm. URL https://developer.nvidia.com/blog/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm/.

[5] Cuda toolkit documentation. URL https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8dc9199943d421bc8bc7f473df12e42.

[6] A. Li. Tartan benchmarks. URL https://github.com/uuudown/Tartan/tree/master/scale-up/scale-up/montecarlo.

[7] NecstLab. Grcuda java benchmarks. URL https://github.com/necst/grcuda/tree/master/projects/resources/java/grcuda-benchmark.

[8] NVIDIA. Cuda toolkit documentation, . URL https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge37112fc1ac88d0f6bab7a945e48760a.

[9] NVIDIA. Nsight systems, . URL https://developer.nvidia.com/nsight-systems.

[10] NVIDIA. Nvprof profiler, . URL https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof.