Mirovia: A Benchmarking Suite for Modern Heterogeneous Computing

Christopher Denny and Sarah Wang

May 10, 2018

Abstract

This paper presents Mirovia, a benchmarking suite developed for modern day heterogeneous computing. Popular benchmarking suites such as Rodinia[1] and SHOC[2] are well written and have many desirable characteristics and features, however they were created years ago when hardware was less powerful and software was less developed. Mirovia is a suite inspired by the aforementioned ones, designed to stress modern architectures, represent a diverse set of application domains in which GPUs dominate, as well as utilize new CUDA features from recent releases. Mirovia contains both existing applications ported from Rodinia and SHOC, as well as newly written applications. The set of benchmarks chosen for the suite provides enough diversity to effectively and comprehensively characterize the strengths and weaknesses of a modern heterogeneous system.

1 Introduction

The use of graphics processing units (GPUs) has become increasingly popular in accelerated computing in industry in recent years. Traditionally, these powerful computational devices were used primarily for 3D game rendering, but their use as General Purpose Graphics Processing Units (GPGPUs) has greatly expanded since then.

Heterogeneous architectures, those using both CPUs and GPUs, are becoming more and more powerful in recent years, and this trend is expected to continue in the future. These systems are also becoming increasingly easier to program with the development of languages such as CUDA and OpenCL. Due to the more widespread availability of heterogeneous systems, these languages are seeing a rise in popularity in the programming world.

Existing benchmarking suites such as Rodinia and SHOC were designed in order to test and better understand the strengths and weaknesses of heterogeneous architectures. A set of applications were curated so that each benchmark exhibits unique behavior that stresses a characteristic or component of the GPU. This is important for programmers and researchers as it helps them pick the most effective hardware and software choices for the task at hand.

This paper makes several key points:

- Presents the areas in which existing benchmarking suites Rodinia and SHOC are lacking, specifically in workload diversity, problem sizes, and programming features.
- Illustrates the improvements made in Mirovia and how these improvements more comprehensively characterize the capabilities of a heterogeneous system
- Analyzes the workloads in Mirovia and demonstrates their diversity in terms of algorithms, applications, and GPU resource usage

2 Related Work

2.1 Rodinia

Rodinia was developed in 2009 as a suite of applications used for benchmarking heterogeneous systems. It consists of real world applications designed to mimic the different Berkeley Dwarfs[3]. These dwarfs are 13 categories of computation that can be used to describe most types of problems. In addition to that, Rodinia benchmarks aim to cover a range of parallel communication patterns and synchronization techniques.

This suite provides an exceptional collection of benchmarks, representing a wide range of dwarfs as well as a wide range of application domains. All of the workloads are considered real world applications, providing valuable information of how a system would perform when used for industry purposes.

However, Rodinia does not provide many performance metrics other than kernel execution time and data transfer time. If a user wanted to run a benchmark several times and get statistics such as mean, median, and standard deviation, they would have to do that by hand. Moreover, due to the fragmented relation between different benchmarks, there's no good unified standard of timing information. At best, some benchmarks (but not all) include statistics like kernel and memcpy times. For other benchmarks, users either have to instrument the code themselves or simply monitor the system time for the overall benchmark. In addition, Rodinia is still using CUDA 4.0. As such it does not use programming constructs or performance improvements from later CUDA versions such as the CUDA event library.

In order to analyze how much each application stresses different components of the GPU, metrics were profiled for each benchmark using nvprof, a command-line version of NVIDIA's Visual Profiler. These metrics quantify utilization as a value from 0 to 10. In this case, utilization for each component indicates how much time a component spent active compared to the total execution time of the kernel. The value 0 means the component was idle, while the value 10 indicates maximum utilization. Since many benchmarks run multiple kernels, the utilization number shown for any component is the maximum utilization seen for the component for any kernel in the benchmark.

Below are two tables, one showing the utilization of the different functional units and one showing the utilization of the memory hierarchy for each application in Rodinia. Since Rodinia does not come with default run parameters but rather a sample run command, the parameters in the sample command were used when running each application.

Benchmark	**Control-Flow**	**Double P.**	**Single P.**	**Half P.**	**Load/Store**	**Special**	**Texture**
Backprop	3	1	6	0	5	3	7
BFS	3	1	6	0	3	2	7
B+ Tree	3	1	6	0	5	3	7
CFDSolver	1	0	4	0	1	1	3
DWT2D	3	9	6	0	5	3	8
Gaussian	3	1	6	0	5	2	7
Heartwall	3	1	6	0	3	2	7
HotSpot	3	1	6	0	3	2	7
Huffman	1	1	4	0	1	1	5
K Means	1	0	4	0	1	1	5
K Nearest Neighbors	3	1	6	0	3	2	7
LavaMD	3	9	6	0	5	2	7
Leukocyte	3	1	6	0	3	2	7
LU Decomposition	3	1	6	0	5	2	7
MummerGPU	3	1	6	0	5	2	7
Myocyte	3	1	6	0	5	2	7
Needleman-Wunsch	3	1	6	0	5	2	7
Particlefilter	1	1	4	0	1	1	5
Pathfinder	1	1	5	0	3	1	5
SRAD	3	1	6	0	5	2	7
Streamcluster	1	1	4	0	1	1	5

Table 1: Functional unit utilization for each benchmark in Rodinia

Benchmark	DRAM	L2	Shared	SysMem	Texture
Backprop	7	4	2	1	6
BFS	7	4	2	1	6
B+ Tree	7	4	2	1	6
CFDSolver	7	4	0	1	6
DWT2D	7	4	2	1	6
Gaussian	7	4	2	1	6
Heartwall	7	4	2	1	6
HotSpot	7	4	2	1	6
Huffman	7	4	1	1	5
K Means	7	4	0	1	5
K Nearest Neighbors	7	4	2	1	6
LavaMD	7	4	2	1	6
Leukocyte	7	4	2	1	6
LU Decomposition	7	4	2	1	6
MummerGPU	7	4	2	1	6
Myocyte	7	4	2	1	6
Needleman-Wunsch	7	4	2	1	6
Particlefilter	7	4	1	1	5
Pathfinder	7	4	2	1	5
SRAD	7	4	2	1	6
Streamcluster	7	4	1	1	5

Table 2: Memory utilization for each benchmark in Rodinia

One interesting observation was that many Rodinia benchmarks had utilization metrics that looked very similar. There were a few exceptions, but most saw around the same levels of utilization for functional units and memory. This is most likely because Rodinia benchmarks are higher level applications, not microbenchmarks that stress specific aspects such as the benchmarks in SHOC. However, the lack of diversity in the utilization numbers for these benchmarks suggest that this metric is not the most informative about how the GPU resources are being used.

2.2 SHOC

Developed in 2010, SHOC is also a benchmarking suite created to test performance in systems with heterogeneous architectures. Unlike Rodinia, SHOC additionally tests for stability through computationally demanding kernels designed to identify devices with bad memory, insufficient cooling, or other problematic components. With regards to performance tests, SHOC divides its benchmarks into level 0 and level 1, where level 0 benchmarks test low-level device characteristics and level 1 benchmarks run basic parallel algorithms. Later, additional level 2 benchmarks were added, which run real world applications similar to the ones in Rodinia.

All SHOC applications are run within a framework that will execute a benchmark for a user-specified number of times and record detailed performance metrics. These metrics include floating point operations per second (FLOPS), bandwidth, kernel time, transfer time, and parity (ratio of transfer to kernel time). The purpose of executing a benchmark several times is to calculate the minimum, maximum, mean, median, and standard deviation for each metric. This framework is extremely useful for evaluating performance. While Rodinia is heavily fragmented, SHOC unites all its benchmarks under this common framework used for configuring runs and recording performance metrics.

In addition, SHOC can be run with predefined problem sizes 1 through 4, each optimized for a different type of architecture:

1. CPUs
2. Mobile/Integrated GPUs
3. Discrete GPUs
4. HPC-Focused or Large Memory GPUs

The biggest drawback with this benchmarking suite is the lack of complex applications. The vast majority of applications in SHOC are considered basic parallel algorithms, which are algorithms that are commonly used as a subroutine of real application kernels. While a variety of dwarfs are represented by these benchmarks, it's difficult to predict how a system would perform in the real world based on these workloads.

Below are tables representing the functional unit and memory utilization for each application in SHOC. All benchmarks were run using the largest preset problem size.

Benchmark	Control-Flow	Double P.	Single P.	Half P.	Load/Store	Special	Texture
BFS	2	0	4	0	1	1	3
DeviceMemory	1	0	9	0	5	2	7
FFT	1	3	3	0	2	1	2
GEMM	1	0	1	0	2	1	1
MaxFlops	1	8	9	0	1	0	1
MD	1	2	2	0	1	1	3
MD5Hash	1	0	8	0	1	1	1
QTC	1	0	1	0	1	1	1
Reduction	1	1	0	1	1	0	2
S3D	1	6	4	0	1	2	4
Scan	1	1	1	0	2	1	2
Sort	2	0	3	0	5	1	4
Spmv	1	1	2	0	1	1	3
Stability	1	1	3	0	2	1	2
Stencil2D	1	1	6	0	2	1	1
Triad	1	0	1	0	1	0	2

Table 3: Functional unit utilization for each benchmark in SHOC)

Benchmark	DRAM	L2	Shared	SysMem	Texture
BFS	1	1	0	1	3
DeviceMemory	7	6	4	1	5
FFT	9	5	2	1	3
GEMM	2	3	3	1	2
MaxFlops	2	1	0	1	1
MD	2	7	0	1	3
MD5Hash	0	1	0	1	0
QTC	3	3	1	1	1
Reduction	6	3	1	1	2
S3D	8	6	0	1	3
Scan	8	4	2	1	3
Sort	8	4	4	1	2
Spmv	4	7	1	1	2
Stability	9	4	2	1	3
Stencil2D	6	4	3	1	2
Triad	2	1	0	1	1

Table 4: Memory utilization for each benchmark in SHOC

Unlike Rodinia, each SHOC benchmark had unique utilization metrics, which shows that these benchmarks are computationally diverse. However, the vast majority of metrics show low utilization, so these benchmarks are not stressing the device as much as desired. In addition, half-precision functional units were introduced with NVIDIA's Pascal GPU, which didn't exist until after Rodinia and SHOC. As a result, none of the benchmarks in either suite utilize the half-precision functional unit.

2.3 Other Benchmarks

While much of this paper focuses on how Mirovia improves on Rodinia and SHOC, there are other benchmarking suites that have been released in recent years[4, 5, 6]. However, while Rodinia and SHOC are still considered general purpose benchmarks, the vast majority of these new suites are skewed towards measuring irregular parallelism performance. As a result Rodinia is still the de facto benchmarking suite used by most research for generic measurements.

3 Motivation

The previous section introduces Rodinia and SHOC, analyzing the pros and cons of each. Here we motivate the creation of Mirovia and outline the key areas in which existing benchmarking suites can be improved.

3.1 GPU Application Domains

The usage of GPUs in industry has expanded to many new domains in recent years. Over time, GPUs have come to dominate a few application domains that are not represented by any workloads in Rodinia or SHOC. The most glaring insufficiency is in the area of machine learning and data analytics. For example, though SHOC does have a neural network benchmark, it is not algorithmically sophisticated and does not represent the current state of the art in machine learning. The same goes for the back propagation application in Rodinia. Mirovia adds applications that are both up-to-date with modern algorithms and allows the suite as a whole to better represent how GPUs are used in industry today. Details about the application domains represented in Mirovia will be provided in the next section.

3.2 Better Dataset Sizes

One of the most obvious aspects of existing benchmarking suites that needs to be updated is dataset sizes. In SHOC, there are 4 preset data sizes that the end user is locked into. This lack of flexibility makes it hard for SHOC to stay relevant in the future, as advancing technology will eventually cause even the largest data size to be too small to stress GPU resources. Meanwhile, Rodinia falls prey to the exact opposite problem, where benchmarks have no preset data size and the user must specify their own problem sizes. Users have to run data generation scripts even if they do not know what input size may be appropriate for the system they are benchmarking.

3.3 Support for Recent CUDA Versions

Another area where Mirovia improves upon existing benchmark suites is support for new features present in recent CUDA releases. In addition to general performance improvements, each new version of CUDA typically releases new programming constructs that can be used to write faster and more efficient code. Multiple such releases have happened in the past few years, yielding features that are not tested at all by other benchmarking suites. Rodinia is currently still using CUDA 4.0, while Mirovia adds support for new features up to CUDA 9.0.

4 Features

This section details the important features in Mirovia as well as the major improvements that have been implemented compared to its predecessors. Mirovia aims to preserve and build upon the desirable features in Rodinia and SHOC while improving aspects of them that are either insufficient or outdated.

4.1 Application Domains and Primitives

While determining a set of benchmarks for Mirovia, consideration was given to both dwarfs as well as application domains. As mentioned before, each of the 13 Berkeley Dwarfs is a low level algorithmic method that characterizes a certain pattern of computation and communication. As a whole, these dwarfs were determined to be representative of real world kernels, and any architecture suited for a broad range of applications should have good performance across all dwarfs. Applications included in Mirovia cover 10 of the 13 dwarves.

Mirovia benchmarks also represent a diverse set of industry applications. As mentioned before, GPUs are now being used for a wide variety of computational tasks in industry, and it's important to include workloads from as many of these domains as possible. A well-rounded heterogeneous system should perform well in all of these domains. The Mirovia benchmarks which run real world GPU applications represent 10 different application domains. The following table shows each benchmark in Mirovia and its respective dwarf and domain, if applicable.

Level	Benchmark	Dwarf	Application Domain
0	BusSpeedDownload	—	—
0	BusSpeedReadback	—	—
0	DeviceMemory	—	—
0	MaxFlops	—	—
1	Breadth First Search	Graph traversal	—
1	General Matrix Multiply	Dense linear algebra	—
1	Pathfinder	Dynamic programming	—
1	Sort	Sorting	—
2	CFDSolver	Unstructured grid	Computational fluid dynamics
2	DWT2D	Spectral method	Image processing
2	K Means	Dense linear algebra	Data mining
2	LavaMD	N-body	Computational chemistry
2	Mandelbrot	—	Numerical analysis
2	Needleman-Wunsch	Dynamic programming	Bioinformatics
2	NeuralNet	—	Machine learning
2	ParticleFilter	Structured grid	Medical imaging
2	SRAD	Structured grid	Computer vision
2	Where	—	Data Analytics

Table 5: Mirovia benchmarks and their respective primitive and application domain

4.2 Performance Metrics

Mirovia modifies the existing framework SHOC uses to collect performance metrics for each benchmark. This framework runs the benchmark several times and measures the mean, median, standard deviation, as well as minimum and maximum for each metric. Metrics that are measured includes kernel time, transfer time, floating point operations per second, and bandwidth (sometimes referred to as rate).

4.3 Dataset Sizes and Data Generation

Mirovia aims to strike a balance between predetermined input sizes available in SHOC and customizable input sizes available in Rodinia. Benchmarks contains preset sizes optimized for systems with different compute capabilities, as well as a mechanism through which users can specify the size and other aspects of their input.

This merges the favorable qualities of both Rodinia and SHOC. While Rodinia allows for users to specify precise data sizes when running benchmarks, it does not give good suggestions as to what sizes would be good for a certain system. There are runfiles present that give sample invocations, but we are still left wanting for concrete sizes that adequately stress a given system. Conversely, SHOC does not allow users to specify custom sizes but does present 4 preset data sizes to choose from when running a benchmark. In Mirovia we merge both of these approaches, letting the user choose at run time which to use. If the user provides a preset size from 1 to 4, the benchmark will use a predetermined input matching the given size. If the user provides benchmark specific arguments(eg for kmeans specifying number of points, dimensionality, etc.) it will generate input matching these options.

There are two places where data generation happens in Mirovia, either statically using a datagen script or dynamically at runtime. Depending on the specific application and the size of the dataset desired, one might be preferred over the other. For applications that require random input, one might wish to generate new input every time the benchmark is run. Otherwise, data can be generated using a script and reused, eliminating the overhead of having to generate new data for each run.

4.4 New CUDA Features

The following section details which CUDA features from recent releases were chosen to be implemented in Mirovia. For each feature, one benchmark was chosen to test it. To test, the benchmark runs twice: one time without the feature and one time with the feature. At the end, the performance metrics from these 2 runs are compared.

HyperQ

HyperQ allows for multiple independent CUDA kernels to run in parallel on the same GPU assuming the resources are available. To speed up execution, GPUs attempt to identify kernels that can be run in parallel. This was a byproduct of old architectures using a single Work Distributor Queue that all kernel calls were appended to. In this model only adjacent calls in the queue could be checked for execution independence. HyperQ uses 32 independent Work Distributor Queues to detect more opportunities for parallel execution. This feature is implemented in Pathfinder, a shortest path algorithm. The runtime for N independent instances executed concurrently using HyperQ is compared to the runtime for N instances executed sequentially.

Dynamic Parallelism

Dynamic Parallelism is the ability to call child CUDA kernels while currently executing a CUDA kernel (nested parallelism). This feature is useful when running algorithms with hierarchical data structures and recursive algorithms with parallelism at each level. Kernels launched through Dynamic Parallelism are completely contained in the parent kernel, meaning that even if only a single thread in a grid launches a child kernel, every thread in the parent must wait for the child to finish before they can exit. To test this feature we added an application that computes a Mandelbrot set to a certain level of precision. This is an appropriate test for evaluating recursive kernel calls because every subset of the Mandelbrot set is a perfect replica of the previous set, allowing for a much simpler and more efficient implementation with Dynamic Parallelism.

Unified Memory

Unified memory is a programming construct that gives the programmer the illusion that the host and the device share an address space. This works by establishing a single memory address space that is visible by all processors in the system. When running applications using this feature, the hardware automatically pages in data needed by a certain processor whenever there's a pagefault.

While this is beneficial to the programmer, using unified memory and having to demand-page data significantly affects the performance of the application. To improve performance, CUDA includes 2 functions the application can call to guide the driver. The first function is cudaMemAdvise(). This function guides the driver by giving "hints" about how certain chunks of memory will be used, such as if certain blocks of memory will be read-only or if certain blocks of memory belong on a specified device (or the host). The second function is cudaMemPrefetchAsync(), and this allows the explicit migration of data to the specified device.

Though using this feature may not always improve performance, the benefit of using this feature is being able to have the host memory act as additional memory to run kernels that need more memory than is available on the device. Unified memory is implemented in BFS. In addition to that, when using graphs that fit in device memory, the runtime of BFS using unified memory is compared to the runtime of that without.

Cooperative Groups

The final CUDA feature implemented in Mirovia is Cooperative Groups, also known as Grid Sync. This feature provides another granularity of synchronization for kernel threads running on a GPU. CUDA has had the __syncthreads() method for some time, a function that synchronizes all the threads in a single block. GridSync takes this to the next level, allowing the user to sync all the threads in the entire grid before beginning next section of computation. This is useful in programs with disjoint phases of computation running right after one another. Cooperative Groups is used in SRAD, where 2 kernels are called back to back with little computation in between.

4.5 Software Design Standards

This suite includes several features and improvements with regards to software design.

Unified Project Hierarchy

While Rodinia is heavily fragmented with each benchmark existing separately, we opted for a more unified approach. One of the issues we ran into when analyzing Rodinia was that every benchmark had its own hand written and customized makefile. This made for widely varying compilation rules, including different optimization levels and cuda compute levels. As mentioned before, every benchmark in Mirova runs within a "harness" which gathers performance metrics, parses benchmark options, and handles input and output. This gives consistency to how each benchmark is run. Though each benchmark may have different options depending on what it does, there are options common to all benchmarks, such as problem size, number of passes, enabling verbose or quiet output, and specifying an input or output file.

CUDA Event API

Mirovia makes use of the CUDA Event API, allowing accurate timing of CUDA functions and kernel calls. This is an improvement from Rodinia, which still uses system time. These events are used to obtain both transfer times during cudaMemcpy() and computation times when a kernel is invoked.

CUDA Error Checking

To improve the user friendliness of the benchmarks, it was necessary to add error checking after important functions. Mirovia implements macros that verifies the success of certain CUDA functions by checking the status code that was returned by it. This error checking mainly done for cudaMalloc() and kernel calls. For example, an out of memory error returned by cudaMalloc() will automatically mean that the kernel will fail. By printing an error message as soon as it happens, the user will be more informed of why the run failed than if the program segmentation faults out of the blue. Though this increases the amount of code sharing between benchmarks, we thought it was a worthwhile trade-off.

GNU Build System (Autotools)

Mirovia uses Autotools, a suite of programming tools designed to make source code easily portable to many Unix-like systems. This ensures that building and running Mirovia is simple and hassle-free regardless of what kind of system it will be running on. Automake will look for the specific libraries required by the project, and will generate Makefiles for each benchmark based on the system's capabilities and how the source files depend on each other. For example, many of the new CUDA features implemented in Mirovia are only available in later CUDA versions, and Autotools allows these benchmarks to be compiled with or without the new feature based on whether or not the CUDA version installed supports it.

Python Driver

The idea of having a script that can run all the benchmarks at once and show output for every test was borrowed from SHOC. SHOC contains a Perl script that runs all the benchmarks and simplifies the output of each benchmark to the names and values of each performance metric taken by the benchmark. This can be anything, such as execution time, bandwidth, or floating point operations per second. In Mirovia, the driver is written in Python. Additionally, the driver allows the user to specify options, such as what subset of benchmarks they want to run, the problem size, the results file location, and more.

5 Benchmarks

In Mirovia, similar to SHOC, benchmarks are divided into levels. These levels represent benchmarks characterizing low level characteristics such as memory bandwidth to performance on real world applications.

5.1 Level 0

Level 0 benchmarks are designed to measure low level characteristics of the hardware. These benchmarks do the simple task of measuring a single capability of the GPU and therefore don't represent any dwarves or application domains.

BusSpeedDownload measures the speed of the PCI bus by repeatedly transferring data of various sizes from the host to the device. The data sizes are varied from 1kb to 500kb.

BusSpeedReadback also measures the speed of the PCI bus, except in the opposite direction. Here, data is transferred from the device to the host.

Both BusSpeedDownload and BusSpeedReadback were included in Mirovia because it is a straightforward test to measure how fast data can be transferred from the host to device and vice versa.

DeviceMemory measures the bandwidth of different components of the memory hierarchy on the device. This includes global, constant, and shared memory.

This benchmark was included in the suite because it tests a basic and low-level component of the device. GPUs are known for hiding memory latency, and because of this, it is important to understand the capabilities of the device memory.

MaxFlops measures the maximum achievable floating point operations per second on the device. In SHOC, this benchmark runs tests using single and double precision.

As mentioned before, there are no benchmarks that utilize the half-precision functional unit, due to this being a fairly recent innovation. This is the perfect benchmark to test the half-precision capabilities of more modern devices. Since only certain systems support these operations, the half-precision test we added may or may not be built and run based on the device's architecture and compute capability.

5.2 Level 1

Level 1 benchmarks are basic parallel algorithms. These algorithms are common tasks in parallel computing and often found in kernels of real applications. While these applications represent a subset of the dwarfs, they are not quite complex enough to represent any application domains. Performance metrics in these workloads include kernel time, transfer time, and may include floating point operations per second or bandwidth.

Breadth First Search (Unified Memory) runs and measures the performance for breadth-first search, a common graph traversal algorithm. This application was included because it is one of the control-flow intensive workloads in the suite.

In addition, this benchmark was chosen to test the unified memory feature in CUDA. This is done by implementing a second version of the benchmark using cudaMallocManaged() instead of cudaMalloc(). Normally, data has to be copied over to the device before the kernel can execute, but using unified memory means that data is paged in on-demand. Therefore, the data transfers are done during the execution of the kernel.

General Matrix Multiply is an application that measures the performance for different types matrix multiplications. The types of matrix multiplications include single and double precision tests with and without transposing the input matrices. This benchmark makes use of the built in matrix multiply function provided by the library libcublas.

General Matrix Multiply was originally slated to be modified to include benchmarking HyperQ performance. However, after analyzing the performance in the Nvidia Visual Profiler, we found that the libcublas library used was simply too efficient at utilizing compute resources for HyperQ to provide a performance benefit. Even a single decently sized matrix multiply saturated the GPU, leaving no room for another multiply to be run in parallel through HyperQ. Meanwhile, once we shrank the problem size down enough for a speedup over sequential execution, the size was so small that any speedup was negligible. For these reasons we chose to instead implement HyperQ in the Pathfinder benchmark.

Pathfinder (HyperQ) Pathfinder is an application that runs a shortest-path algorithm. This is one of the benchmarks present that serves as a test of irregular parallelism. While most conventional parallel algorithms have uniform behavior across the different threads, irregular algorithms are characterized by different threads performing different executions. Pathfinder is a good example of this because each thread's execution depends on which node it is currently processing. Depending on graph connectedness and general structure, different threads can experience highly variable behavior. In addition to this, pathfinder will experience much higher control flow unit utilization compared to regular parallelism algorithms as each thread needs to decide how to execute independently.

HyperQ is most beneficial when there are multiple independent kernels operating on disjoint data. To simulate this in Pathfinder, we simply run multiple instances of the pathfinder kernel in parallel on systems that support HyperQ (CUDA 5.0 and later). More concretely, we run 32 instances, each in their own CUDA stream.

Sort is an application that runs a radix sort on an array of integers. This application was originally intended to test the HyperQ feature in CUDA, but since the sort is done in several stages using multiple kernel calls, the implementation of HyperQ would have been more complex than necessary. Performance metrics for this application include kernel time, transfer time, as well as sorting rate in terms of the amount of data sorted per second.

5.3 Level 2

Level 2 benchmarks are real world application kernels. Benchmarks in this level are applications that can be found in industry, and therefore represent a variety of GPU application domains. Since it's difficult to pinpoint the exact number of floating point operations done during kernel execution, performance metrics in this level are usually limited to kernel time, transfer time, and parity (ratio of transfer to kernel time).

CFD Solver stands for computational fluid dynamics, an area in which GPU-accelerated computing dominates. This application solves the three-dimensional Euler equations for compressible flow. This workload optimizes effective GPU memory bandwidth by reducing total global memory accesses and overlapping redundant computation.

GPUDWT stands for discrete wavelet transform, an image and video compression algorithm that is also a popularly used digital signal processing technique. This benchmark implements both forward and reverse, as well as 9/7 and 5/3 transforms. The 9/7 transform uses floats while the 5/3 transform uses integers, so it's important to measure the performance for both.

KMeans is a popular clustering algorithm used in data mining. This algorithm is fairly simple and shows a high degree of data parallelism. At the beginning, K centers are chosen. In each iteration, each data point is assigned to a center, and at the end of each iteration, each center is recomputed as the mean of all the data points in its cluster until the two converge.

LavaMD is a benchmark that calculates n body particle interaction. The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since ones at larger distances exert negligible forces. Thus the box size s chosen so that cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes.

Mandelbrot (Dynamic Parallelism) computes an image of a Mandelbrot fractal, a self repeating geometric pattern that loops back on itself at ever decreasing sizes. One common algorithm to calculate these images in the past was the Escape Time Algorithm, which calculates the value for different pixels on a per pixel basis.

This benchmark was added specifically to test Dynamic Parallelism, a feature added to CUDA in version 5.0. On systems supporting Dynamic Parallelism, the benchmark switches to using the Mariani-Silver Algorithm. Unlike Escape Time, this procedure starts out coarse grained, and only iterates at a finer resolution if necessary for certain subsections. This behavior is perfectly suited for Dynamic Parallelism, with kernels only recursively calling themselves for sections that require further precision.

Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment.

NeuralNet is an application that runs a subset of kernels involved in AlexNet, a convolutional neural network (cNN) originally written for the purpose of image recognition and classification.

As mentioned earlier, SHOC and Rodinia both have neural network applications, but these benchmarks are algorithmically too simple and outdated to represent the current state-of-the-art in machine learning. Convolutional neural networks are currently very popular in this application domain. More specifically, these types of neural networks use batch processes. By transforming input data into a matrix, operations can be done to the entire dataset at once. Batch processes allow any algorithm to be run on the entire dataset without having to focus on individual examples, therefore greatly increasing efficiency.

ParticleFilter s a statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework. The PF has a plethora of applications ranging from video surveillance in the form of tracking vehicles, cells and faces to video compression. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells.

After selecting the target object, the PF begins tracking it by making a series of guesses about the current frame given what is already known from the previous frame. The PF then determines the likelihood of each of those guesses occurring using a predefined likelihood model. Afterwards, the PF normalizes those guesses based on their likelihoods and then sums the normalized guesses to determine the object's current location. Finally, the PF updates the guesses based on the current location of the object before repeating this process for all remaining frames in the video.

SRAD (Cooperative Groups) is a computer vision application used for reducing noise, or "speckles", in images without destroying important image features. This is done using partial differential equations.

Since each stage of this application operates on the entire image, SRAD requires synchronization after each stage. This makes SRAD the ideal benchmark to test the performance of using cooperative groups in CUDA. Instead of making 2 separate kernel calls (one for each stage), the version of the benchmark using cooperative groups launches a cooperative kernel using cudaLaunchCooperativeKernel(), allowing for SRAD to be completed using a single kernel call. In this kernel, threads are synced across the entire grid before proceeding onto the next stage.

Where is a new benchmark written for Mirovia involving relational algebra. GPUs are becoming increasingly popular for data analytics because relational algebra operations are easy to parallelize. This benchmark essentially acts like a filter for a set of records, returning a subset of the input records based on whether or not each record fulfills a set of conditions. It first maps each entry to a 1 or a 0, before running a prefix sum and using both of these auxiliary data structures to reduce the input data to just the matching entries. Though Rodinia has a SQLLite application in its latest release, it was not included in Mirovia because we thought a workload that does filtering would be better suited to represent the relational algebra application domain.

6 Evaluation

In this section, we analyze the applications in Mirovia in terms of runtime characteristics, diversity, and performance. We ran and tested our benchmarks on a machine with the following specifications:

- Ubuntu 16.04.3 LTS
- Linux 4.4.0-98-generic
- CPU: Intel(R) Xeon(R) E5-2650 v4
  - 12 Core, 24 Thread
  - 2.20Ghz Base Freq, 2.90Ghz Turbo Freq
  - 256K L2 Cache
  - 30M L3 Cache
- GPU: Nvidia Tesla P100-SXM2
  - Cuda 384.90
  - 16GB HBM2 Memory
  - 1328Mhz SM clock speed
  - 715Mhz Memory clock
- 126GB DDR4 Memory

6.1 Performance Analysis

It was mentioned earlier that architectures are a lot more powerful now than when Rodinia and SHOC were first released. Because of this, many of the existing applications do not stress GPU resources on more modern systems as much as they should. These applications in Mirovia were modified in order to utilize more of the available resources.

To evaluate the effectiveness of our modifications, we used the NVIDIA command-line profiler to collect metrics on the utilization of the different functional units on the device, as well as the different components of the memory hierarchy.

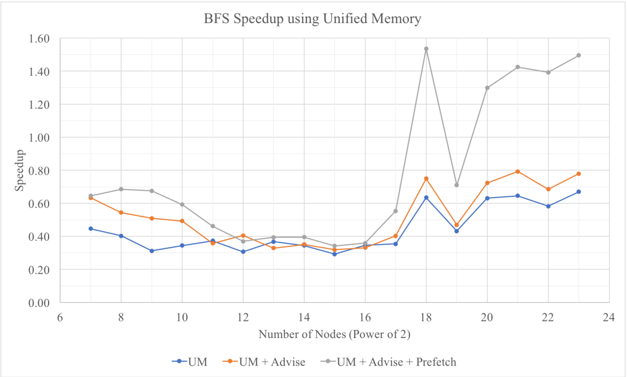
Utilization is measured as the amount of time a component spent active divided by the total execution time of the application. The more time a component spends active, the higher the utilization will be. The NVIDIA profiler provides utilization on a scale from 0-10, where 0 means the component was idle and 10 means the component was at maximum utilization. Here, we show the utilization of the different functional units as well as device memory hierarchies in Mirovia.

The bar graphs show a good spread of utilization over the different functional units and categories of the memory hierarchy. General Matrix Multiply runs both single and double precision tests and thus we see full utilization of both these functional units. Meanwhile various benchmarks all make use of different memory components such as texture and shared memory. Gemm and Sort stand out as examples that utilize shared memory for fast access to sections of elements by multiple threads in their respective algorithms. Meanwhile Particlefilter sees higher use of texture memory, which follows given that spatially localized threads analyze particles within a localized zone. One especially important utilization graph is Maxflops. This microbenchmark shows high utilization of the half precision functional unit. This is especially important since both Rodinia and SHOC are old enough that half precision did not exist when they came out.

6.2 CUDA Feature Analysis

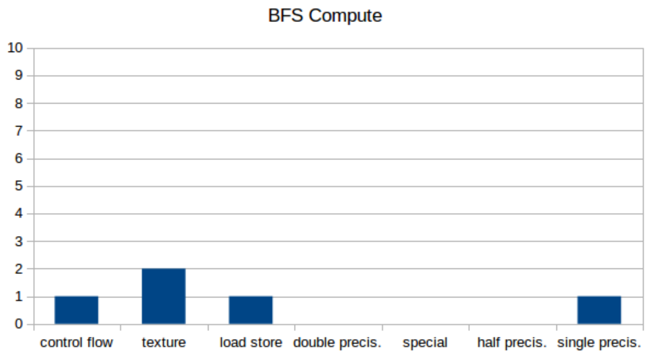
In addition, we analyzed benchmarks that implement new CUDA features to find out how the use of the feature impacts the performance of the application. To do this, we looked at the speedup of the application using the feature over various problem sizes.

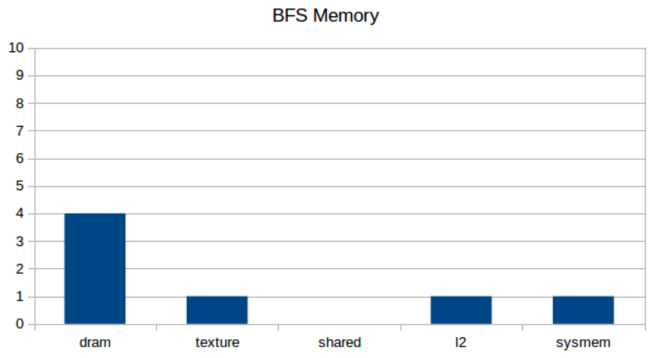
Unified Memory

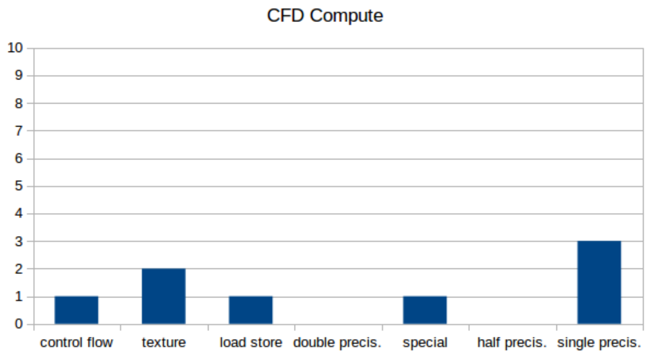


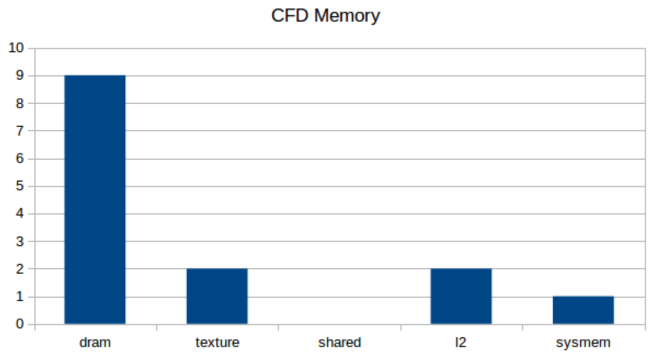
For this feature, the kernel time plus the transfer time of BFS without unified memory was compared to the kernel time of BFS with unified memory, since there is no explicit transfer time when using unified memory. Three different versions of BFS using unified memory were tested and each was compared to a version of BFS that doesn't utilize any new features.

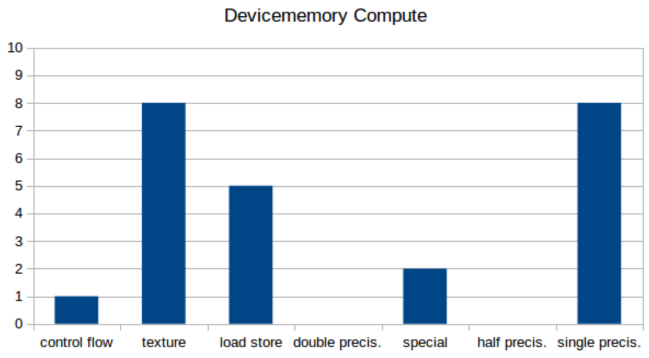
The first version was a version that uses unified

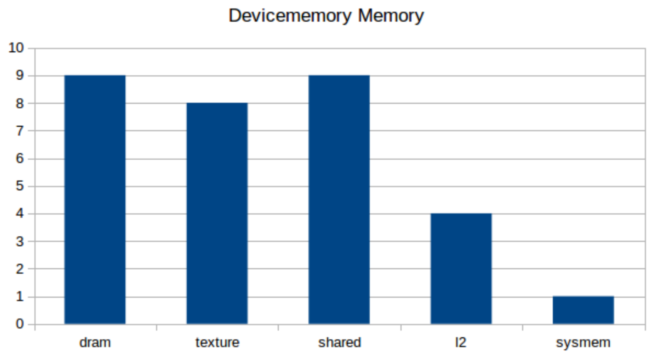

BFS Compute

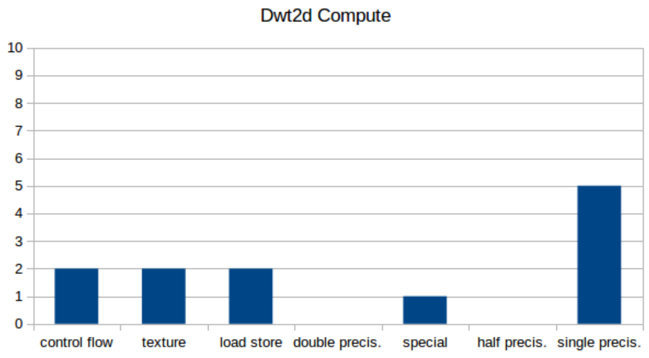

BFS Memory

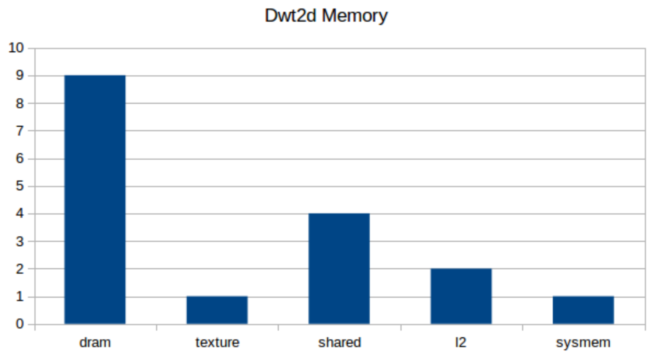

CFD Compute

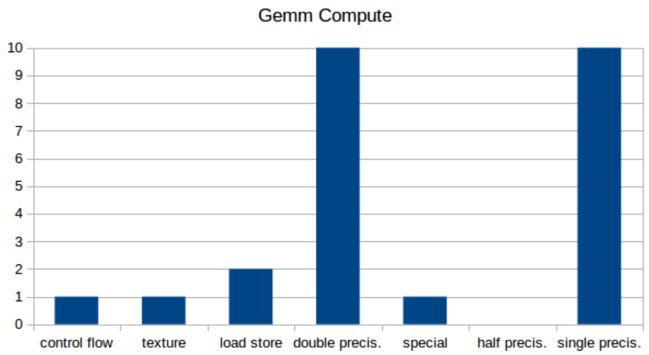

CFD Memory

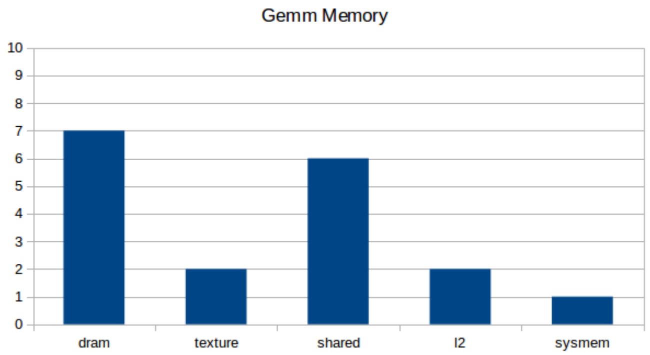

Devicememory Compute

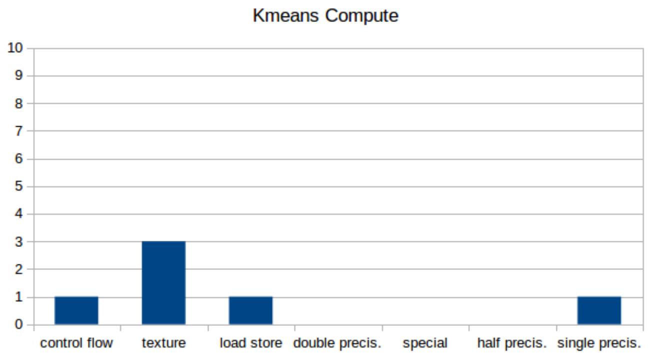

Devicememory Memory

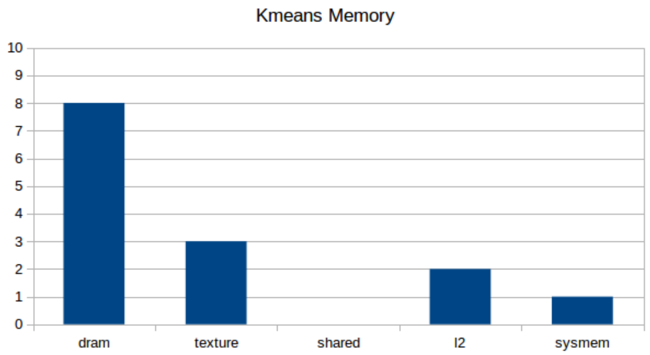

Dwt2d Compute

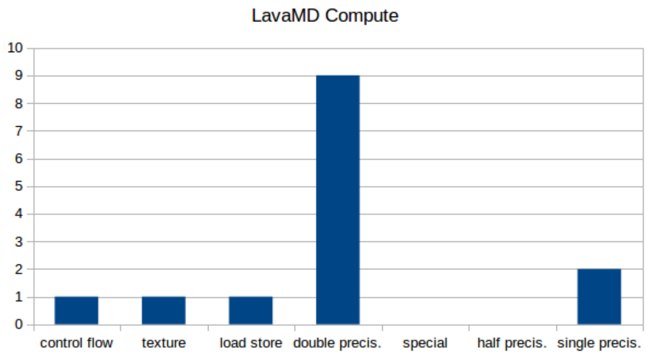

Dwt2d Memory

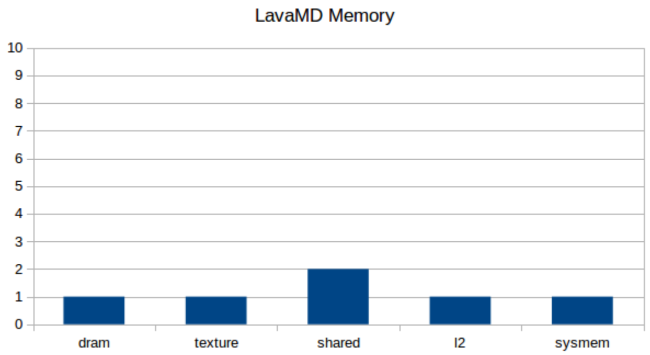

Gemm Compute

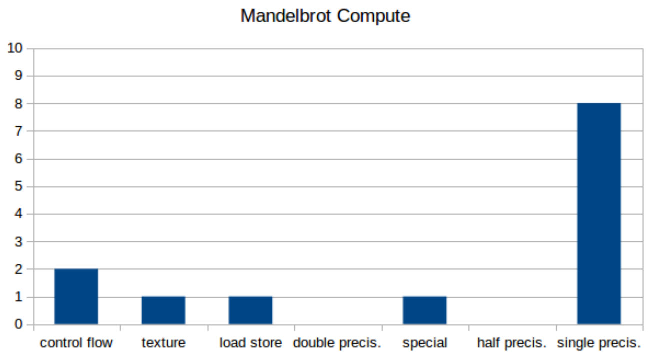

Gemm Memory

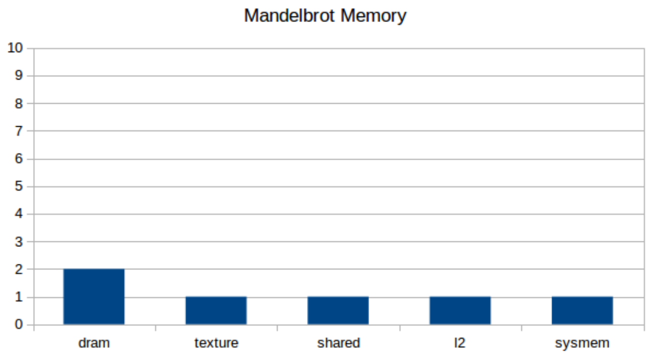

Kmeans Compute

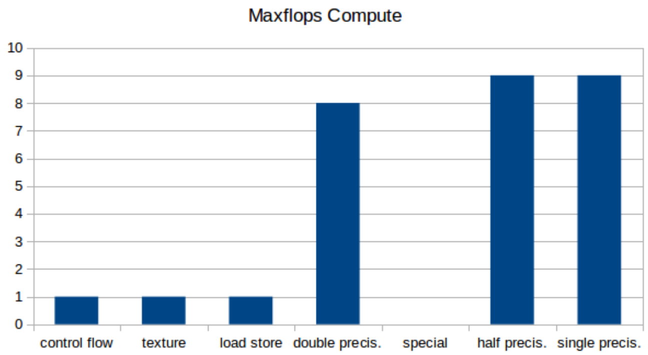

Kmeans Memory

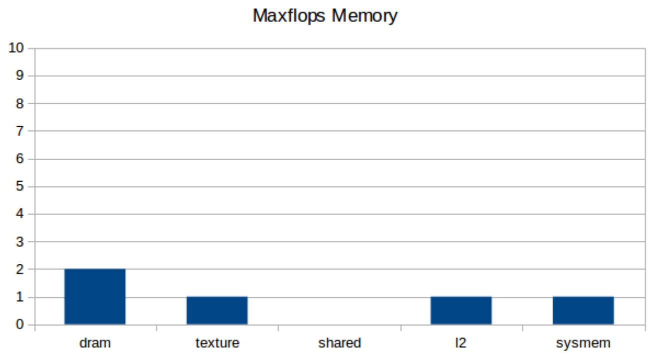

LavaMD Compute

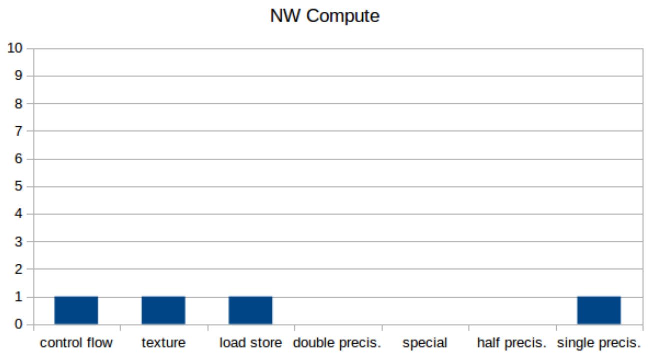

LavaMD Memory

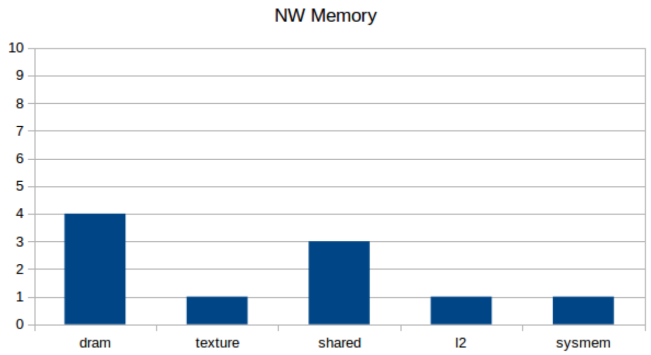

Mandelbrot Compute

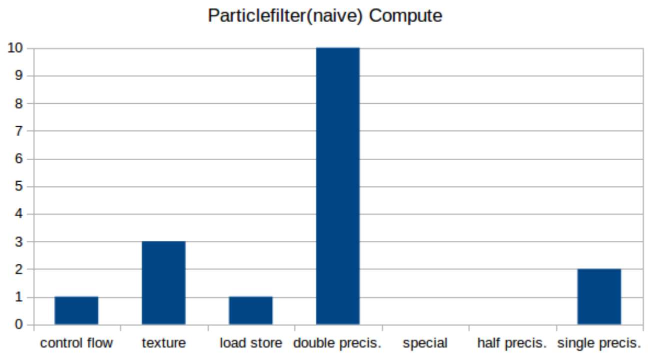

Mandelbrot Memory

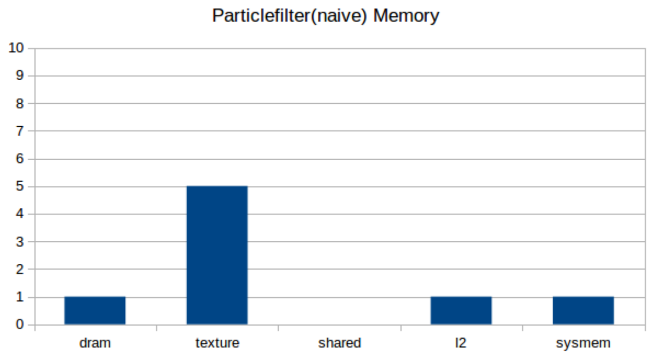

Maxflops Compute
0
1
2
3
4
5
6
7
8
9
10
control flow
texture
load store
double precis.
special
half precis.
single precis.

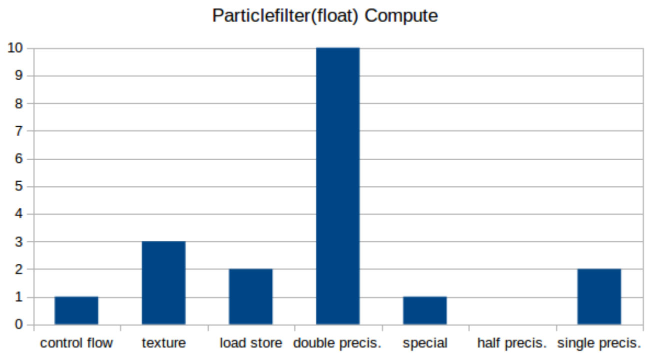

Maxflops Memory
0
1
2
3
4
5
6
7
8
9
10
dram
texture
shared
l2
sysmem

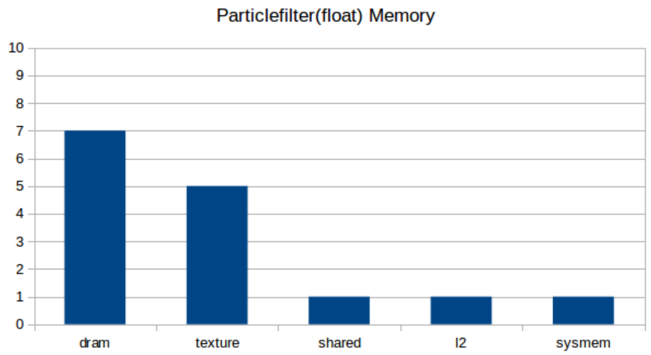

NW Compute
0
1
2
3
4
5
6
7
8
9
10
control flow
texture
load store
double precis.
special
half precis.
single precis.

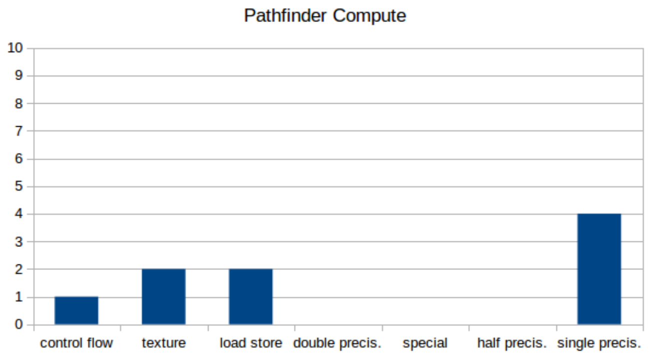

NW Memory
0
1
2
3
4
5
6
7
8
9
10
dram
texture
shared
l2
sysmem

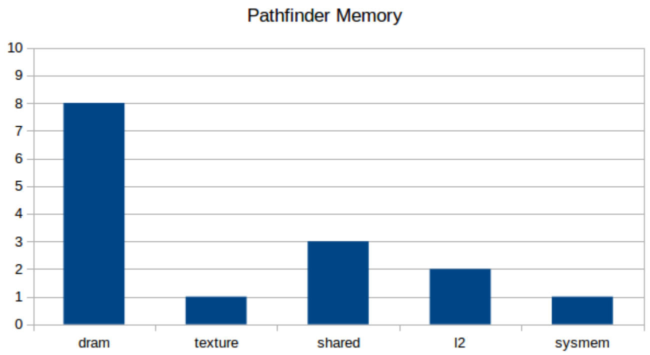

Particlefilter(naive) Compute
0
1
2
3
4
5
6
7
8
9
10
control flow
texture
load store
double precis.
special
half precis.
single precis.

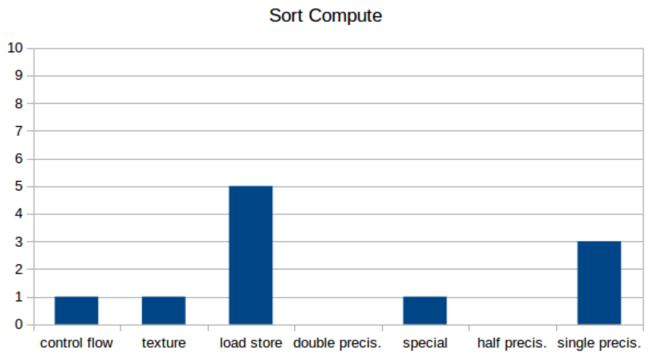

Particlefilter(naive) Memory
0
1
2
3
4
5
6
7
8
9
10
dram
texture
shared
l2
sysmem

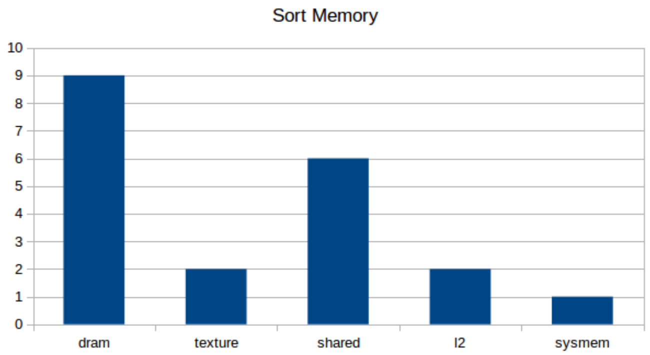

Particlefilter(float) Compute
0
1
2
3
4
5
6
7
8
9
10
control flow
texture
load store
double precis.
special
half precis.
single precis.

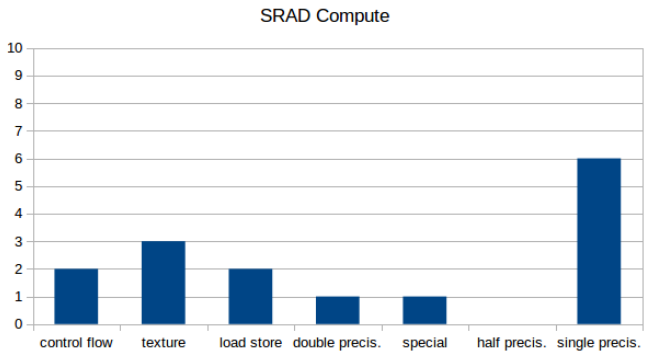

Particlefilter(float) Memory
0
1
2
3
4
5
6
7
8
9
10
dram
texture
shared
l2
sysmem

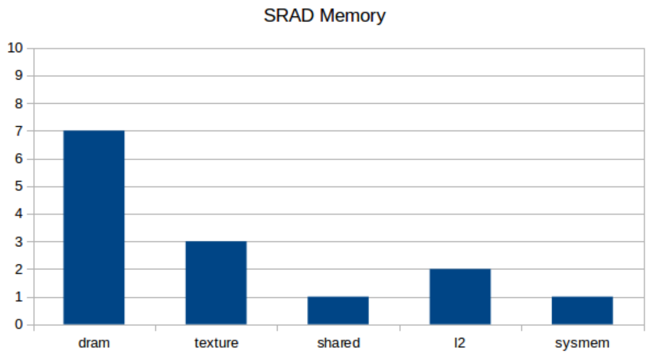

Pathfinder Compute

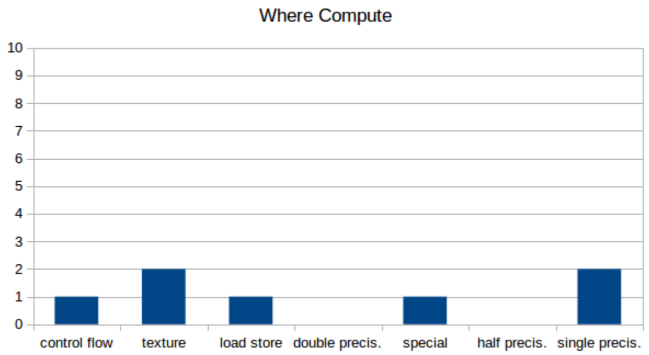

Pathfinder Memory

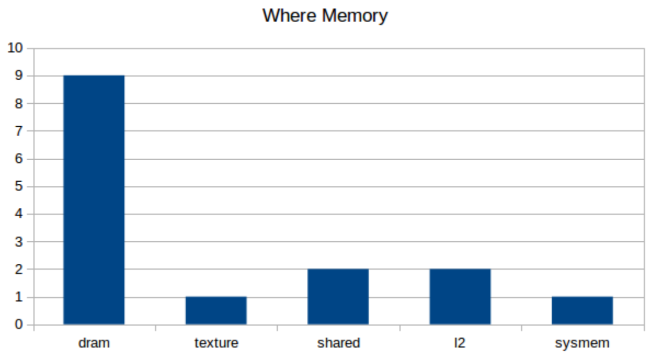

Sort Compute


Sort Memory


SRAD Compute


SRAD Memory


Where Compute

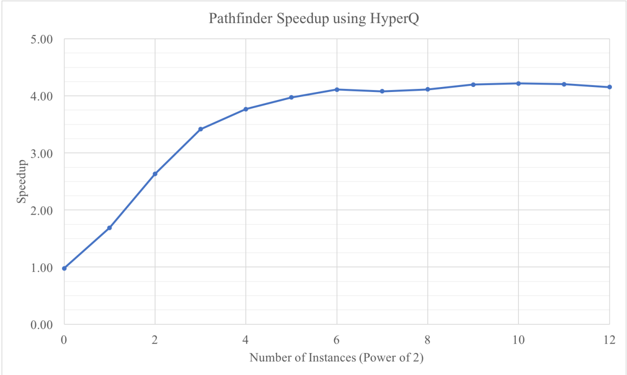

Where Memory

memory but neither cudaMemAdvise() nor cudaMemPrefetchAsync(), two functions that were introduced earlier in this paper. The second version uses unified memory with only cudaMemAdvise(), and the last version uses unified memory with both cudaMemAdvise() and cudaMemPrefetchAsync(). Only when prefetching was introduced was BFS with unified memory able to run faster than the baseline version.

Additionally, the speedup was fairly inconsistent and did not exactly scale with the input size. This is because the execution path is highly dependent on the graph that is generated. Since data is randomly generated, this introduces an element of randomness to the speedup over various problem sizes.

Though unified memory may not provide performance improvements, it simplifies the code for the programmer. Instead of having to allocate memory on both the host and the device and having to explicitly transfer the data back and forth, unified memory allows the programmer to pretend like the host and the device share a single address space.

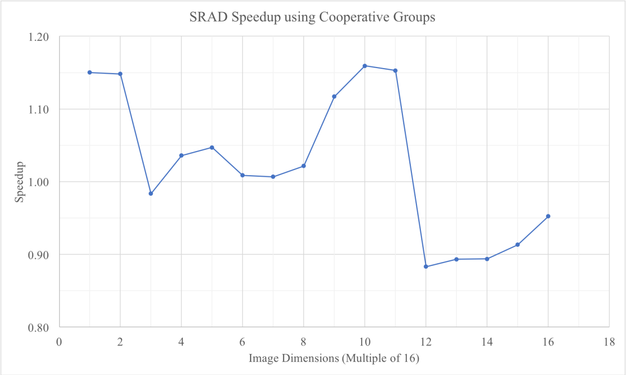
HyperQ



HyperQ was added to the level 1 Pathfinder benchmark. Since this CUDA feature increases the utilization when multiple independent kernel can be run concurrently, we just ran multiple instances of Pathfinder on different streams. The graph above shows the relative speedup as the number of concurrent pathfinder kernels increases. When measuring the comparative speedup of HyperQ we chose not to include transfer time since that would stay the same regardless.

As shown above, the speedup gained from HyperQ increases as the number of parallel kernels scales up. This speedup levels out around 32 parallel instances. This makes sense as at this point the benchmark is saturating all 32 independent work queues. In addition to this we see speedup starting at a little under 1x for a single instance, and ramping up to 4x from there. This follows as increasing the number of instances here makes use of more of the work queues that are just sitting idle.

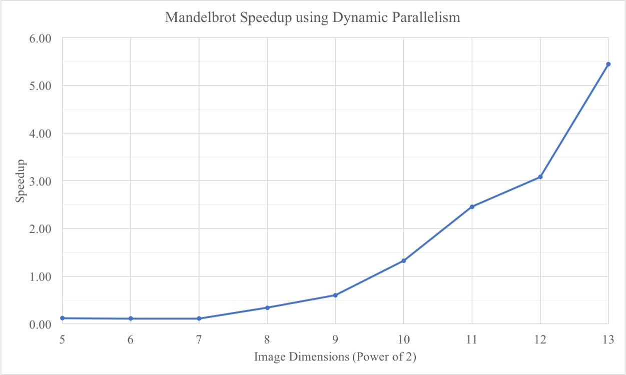
Cooperative Groups



Similar to HyperQ, the kernel time for SRAD using a cooperative kernel was compared to the kernel time for the original SRAD implementation. In this case, the transfer time was not included because it would have been the same for both.

The biggest drawback of using cooperative groups is the limit on how many blocks can be launched. Because of this, SRAD using a cooperative kernel could not be run on image sizes greater than 256x256. Therefore, to get more data points, we varied the problem size by multiples of 16 instead of powers of 2.

Dynamic Parallelism



For this feature, the speedup was measured using the kernel times for Mandelbrot with and without Dynamic Parallelism. Like most of the other features, transfer time was not included in the speedup because it was the same for both versions of the benchmark. This benchmark shows one of the cleanest increases in speedup as problem sizes increase. This primarily comes down to the efficiency of the two algorithms used and what Dynamic Parallelism allows. While the traditional Escape Time algorithm is forced to calculate values for every pixel, Mariani Silver is allowed to subdivide and thus ignore every increasing swaths of the image. This is shown by the ever increasing speedup as image size increases.

While this benchmark's inclusion of dynamic parallelism is used for regular parallelism, this feature can also be used for more varied implementations. The mandelbrot kernel explicitly calls itself over and over on smaller patches of the image. Other programs may choose to have a master kernel that calls multiple diverse subkernels. Mandelbrot was still enough to stress Dynamic Parallelism and is thus what we chose to include in Mirovia.

7 Individual Contributions

7.1 Benchmarks

During the development of Mirovia, these were our contributions for each benchmark. Overall, the workload was split evenly between us, given that certain tasks, such as implementing new benchmarks or CUDA features, are significantly more difficult and time consuming than tasks such as porting a benchmark from another suite or writing a data generation script.

Benchmark	**Ported from Rodinia/SHOC**	**Implemented as new**	**Wrote static/dynamic data generation**	**Implemented CUDA feature**
BusSpeedDownload	S	—	—	—
BusSpeedReadback	S	—	—	—
DeviceMemory	S	—	—	—
MaxFlops	S	—	—	—
Breadth First Search	S	—	S	S
General Matrix Multiply	S	—	S	—
Pathfinder	S	—	S	C
Sort	C	—	C	—
CFDSolver	C	—	C	—
DWT2D	S	—	S	—
K Means	S	—	S	—
LavaMD	C	—	C	—
Mandelbrot	—	C	C	C
Needleman-Wunsch	S	—	S	—
NeuralNet	—	S	S	—
ParticleFilter (Naive)	C	—	C	—
ParticleFilter (Float)	C	—	C	—
SRAD	C	—	C	S
Where	—	C	C	—

Table 6: Individual contributions to each benchmark

7.2 Sarah

Additionally, Sarah made the following contributions:

- Set up the GNU build system using autotools. This automatically configures and builds the project according to the specs of the system the repository is cloned onto

- Wrote the Python driver for running the benchmarking suite. This allows the user to run the entire suite of workloads with a single command.

- Modified the framework in which benchmarks run, including performance metrics output, run options, as well as using and writing input and output files.

7.3 Christopher

Additionally, Christopher made the following contributions:

- Implemented the feature using autotools that determines whether or not to build CUDA features based on the CUDA version installed on the system

- Modified the build system to separate compilation and linkage of CUDA device code for certain benchmarks which use dynamic parallelism

- Analyzed each benchmark using metrics from the NVIDIA command-line profiler to adjust preset problem sizes

8 Future Work

8.1 Analysis of Existing Suites

As mentioned before, our analysis of Rodinia and SHOC using metrics from NVIDIA's command-line profiler loses meaning because many of the Rodinia benchmarks showed the same numbers for memory utilization. To improve this, other useful numbers can be looked at, including throughput, as well as efficiency, which is the ratio of realized throughput to maximum achievable throughput.

8.2 GPUDirect RDMA

One future goal is to add benchmark support for GPUDirect RDMA. This is a feature that allows for direct data exchange between devices on a PCI bus. This is a form of RDMA (Remote Direct Memory Access) that allows other devices (e.g. other GPUs, network adapters) to directly write to device memory. Using this feature greatly reduces the time required for data transfers by bypassing typical memory copies across various data planes. This is especially useful for pipelined data processing workloads where there are multiple disjoint transformations acting on the same data. Typically these workloads involve copying data from host to device, running a program, and transferring data back for every stage of the pipeline. GPUDirect RDMA allows for data to stay off host the entire time, moving directly from stage to stage of the workload.

8.3 Further Optimization

One aspect that can always be improved on is further optimizing different benchmarks to take advantage of existing architectures and features. Specifically, in the future we would like to convert more benchmarks to make use of constant memory, texture memory, shared memory, etc for continued performance gains. These changes are not one and done but rather constant iterative improvements that can be continually applied for the life of the benchmarking suite.

9 Conclusion

In creating Mirovia we aimed to modernize aspects of popular existing suites such as Rodinia and SHOC. To do this, we improved the spread of benchmarks included, bringing in new programs from different domains, while also adapting problem sizes to the abilities of modern hardware. Most importantly, we added support for measuring the performance of many new features that were introduced in recent years. Features like Cooperative Groups, HyperQ, and half precision arithmetic are all new enough that no current suite tests them at all. In this way we present Mirovia as a more complete benchmarking suite for the modern era.

References

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[2] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76. IEEE, 2009.

[5] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195. IEEE, 2013.

[6] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55. IEEE, 2015.