NUPAR: A Benchmark Suite for Modern GPU Architectures

Yash Ukidave*, Fanny Nina Paravecino*, Leiming Yu*, Charu Kalra*, Amir Momeni*,
Zhongliang Chen*, Nick Materise*, Brett Daley*, Perhaad Mistry†and David Kaeli*
*Electrical and Computer Engineering, Northeastern University, Boston, MA
†Advanced Micro Devices Inc. (AMD), Boxborough, MA
{yukidave, fninaparavecino, ylm, ckalra, amomeni, zhonchen, nmaterise, bdaley,
kaeli}@ece.neu.edu, Perhaad.Mistry@amd.com

ABSTRACT

Heterogeneous systems consisting of multi-core CPUs, Graphics Processing Units (GPUs) and many-core accelerators have gained widespread use by application developers and data-center platform developers. Modern day heterogeneous systems have evolved to include advanced hardware and software features to support a spectrum of application patterns. Heterogeneous programming frameworks such as CUDA, OpenCL, and OpenACC have all introduced new interfaces to enable developers to utilize new features on these platforms. In emerging applications, performance optimization is not only limited to effectively exploiting data-level parallelism, but includes leveraging new degrees of concurrency and parallelism to accelerate the entire application.

To aid hardware architects and application developers in effectively tuning performance on GPUs, we have developed the *NUPAR* benchmark suite. The *NUPAR* applications belong to a number of different scientific and commercial computing domains. These benchmarks exhibit a range of GPU computing characteristics that consider memory-bandwidth limitations, device occupancy and resource utilization, synchronization latency and device-specific compute optimizations. The *NUPAR* applications are specifically designed to stress new hardware and software features that include: nested parallelism, concurrent kernel execution, shared host-device memory and new instructions for precise computation and data movement. In this paper, we focus our discussion on applications developed in CUDA and OpenCL, and focus on high-end server class GPUs. We describe these benchmarks and evaluate their interaction with different architectural features on a GPU. Our evaluation examines the behavior of the advanced hardware features on recently-released GPU architectures.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Benchmarking*

General Terms

Profiling, Performance Measurement, Benchmarking,

Keywords

Benchmark suite, GPUs, CUDA, OpenCL

1. INTRODUCTION

Heterogeneous computing using accelerators such as multi-core CPUs, GPUs, and FPGAs has gained a lot of traction in recent years. The programmability and performance of the GPUs have increased to support a range of throughput-oriented workloads belonging to various scientific domains. The availability of mature software frameworks has helped GPUs become a commonly used device in throughput computing.

In more recent accelerator platforms we are beginning to see applications that come with stricter timing constraints, stringent resource requirements, opportunities for concurrent execution and irregular memory access patterns. Many of these applications have been successfully moved to GPU platforms [33]. Programming frameworks such as CUDA, OpenCL, and OpenACC have introduced new features that address many of these challenges on GPUs. Modern GPU architectures constantly evolve by adding support for such advanced constructs introduced in the programming frameworks. The performance of applications on GPUs can improve dramatically if these programming features are used efficiently. It is essential to provide researchers with a proper set of benchmarks that can appropriately exercise such advanced features, to consider hardware and software performance tradeoffs on different GPU platforms, and to identify performance bottlenecks and evaluate potential solutions.

Researchers have developed a number of benchmark suites to study different aspects of a GPU architecture [7, 10, 16, 31]. The goals of a benchmark suite can vary depending on the class of system they target. As GPU systems have evolved, older benchmark suites gradually become less relevant. This evolution calls for the development of a new generation of GPU benchmarks that target modern GPU architectures using advanced programming frameworks.

In this paper, we provide *NUPAR*, a novel benchmark suite to equip architects and application designers with

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2688046.

appropriate workloads to evaluate the performance of the emerging class of GPUs. We incorporate a set of real-world applications that appropriately exercise advanced architectural features of the GPUs. The applications have been developed using features from CUDA and OpenCL frameworks [1, 17]. The *NUPAR* suite provides eight different applications spanning seven different computing domains. The nature of computation for each application is different, providing a mix of behaviors. The properties of the applications include real-time constraints, memory-bound operations, compute-bound execution, heavy synchronization, and concurrent execution. Each application highlights one or more advanced architectural features of the GPU according to the nature of their computation. The application kernels are optimized to better utilize the GPU architecture.

The applications written in CUDA can be easily ported to OpenCL, and vice versa. CUDA and OpenCL programming models are quite similar to each other in many respects. The OpenCL model is based on a Compute Device that consists of Compute Units with Processing Elements. These Compute Units are equivalent to CUDA's Streaming Multiprocessors, which contain CUDA cores. In OpenCL, a host program launches kernel with *work-items* (vs. *threads* in CUDA) over an index space. These work-items are further grouped into *work-groups* (vs. *thread blocks* in CUDA). Furthermore, both have similar device memory hierarchies abstracted into different address spaces. However, CUDA is currently supported on NVIDIA GPUs and multicore CPUs [32], whereas OpenCL is supported on many different heterogeneous devices including many GPUs, multi-core CPUs and FPGAs.

We evaluate the CUDA workloads on an NVIDIA K40 and OpenCL workloads on an AMD Radeon HD 7970 GPU. We measure speedup against a GPU-accelerated baseline that is unoptimized. We use NVIDIA's Visual Profiler [23] for profiling CUDA applications and AMD's CodeXL [3] for profiling OpenCL applications.

The major contributions of this paper include:

- we introduce a new benchmark suite to study advanced architectural features of modern GPUs,
- we provide benchmarks that can cover a wide range of computation models, exercising properties common in emerging heterogeneous applications, and
- we utilize these benchmarks to exercise advanced architectural features on GPUs to illustrate their impact on the performance of applications.

The remainder of this paper is organized as follows. Section 2 covers the motivation for our work. In Section 3, we describe the architectural and programming features that we are targeting in our applications. Section 4 introduces the applications chosen from various domains that are included in *NUPAR*. Section 5 discusses our evaluation methodology. In Section 6, we discuss experimental results using the benchmarks. Section 7 provides a comparison between different programming frameworks. Section 8 surveys related work and Section 9 concludes the paper and considers directions for future work.

2. MOTIVATION

NVIDIA Devices	NVIDIA C2070	NVIDIA K40
Microarchitecture	Fermi	Kepler
Fabrication	40nm	28nm
Compute Capability	2.0	3.5
CUDA Cores	448	2880
Core Frequency	575 MHz	745 MHz
Memory Bandwidth	144 GB/s	288GB/s
Peak Single Precision FLOPS	1288 GFlops	4290 GFlops
Peak Double Precision FLOPS	515.2 GFlops	1430 GFlops

Table 1: Evolution of NVIDIA GPU architecture from Fermi to Kepler.

AMD Devices	Radeon HD 5870	Radeon HD 7970
Microarchitecture	Evergreen	Southern Islands
Fabrication	40nm	28nm
Stream Cores	320	2048
Compute Units	20	32
Core Frequency	850 MHz	925 MHz
Memory Bandwidth	153.6 GB/s	264 GB/s
Peak Single Precision FLOPS	2720 GFlops	3789 GFlops
Peak Double Precision FLOPS	544 GFlops	947 GFlops

Table 2: Evolution of AMD GPU architecture from Evergreen to Southern Islands.

GPUs have increased in performance, architectural complexity and programmability over the years. This trend can be observed in various generations of GPUs from different vendors [21]. Table 1 shows how NVIDIA GPUs have improved from Fermi to Kepler microarchitecture in terms of their compute capability, architectural complexity, and memory bandwidth. Table 2 shows similar improvements that have been made in AMD's Southern Islands generation of GPUs over its predecessor. Not only have the GPUs evolved in terms of sophistication and capabilities, but the programming frameworks have evolved dramatically to support the hardware changes in the GPU architectures.

Previous benchmark suites have been developed specifically to understand the performance of the GPU devices. The Rodinia suite provides a set of GPU programs targeting multi-core CPU and GPU platforms based on the *Berkeley dwarf* taxonomy [7]. Rodinia benchmarks highlight architectural support for memory-bandwidth, synchronization and power consumption. Another important GPU benchmark suite is the Parboil suite. Parboil provides GPU workloads which exercise architectural features of GPUs such as floating point throughput, computational latency and cache effectiveness [31]. Both of these benchmark suites have served the GPU architectural research community well over many years. In contrast to Rodinia and Parboil, the Scalable Heterogeneous Computing (SHOC) [10] benchmark suite provides a range of low-level benchmarks based on scientific computing workloads. The Valar benchmark suite emphasizes the host-device interaction on Accelerated Processing Units (APUs) and between multi-core CPUs and discrete GPUs [16]. The workloads provided in the Valar suite utilize multiple command-queue/streams features of a true heterogeneous programming framework that combines CPUs and GPUs.

Unfortunately, the available GPU benchmark suites do not provide that can applications to properly stress the latest architectural features appearing on GPUs. Some of these features include concurrent kernel execution, dynamic parallelism, unified memory hardware, improved double precision and atomic instructions. Many of these features are already supported in the new CUDA-6 and OpenCL 2.0 standards. These advances in runtime systems and architectures have motivated us to assemble a new benchmark suite that focuses attention on the latest architectural features on GPUs. These real world applications belong to a diverse set of domains and perform computations to stress different combinations of these new features. This allows us to study the impact of each architectural feature. Architects are no longer fully responsible for identifying why a particular benchmark achieves a larger advantage than another benchmark when studying a particular architectural feature. We have identified the characteristics of each benchmark, and users can leverage this guidance as they explore the benefits of each new feature.

3. USE OF ARCHITECTURAL AND PROGRAMMING FEATURES FOR OPTIMIZATION

As mentioned in Section 2, today's programming frameworks and architectures have rapidly evolved to enable new programming patterns and improve performance. In this section, we describe some of the new architectural and programming features that we target with our applications. Traditionally, optimizations applied to GPU workloads consist of effective memory management techniques and use of sophisticated algorithms to leverage the parallelism offered by GPUs. As new architectural features are being delivered to the market, it is important that application developers understand the nature of these features so that they can reap the full benefits of them when developing their applications.

The number of features appearing in new GPU devices go well beyond the few that are summarized in this section. Most of these features are available on both CUDA and OpenCL frameworks using similar programming nomenclature.

3.1 Nested Parallelism

Nested parallelism is defined as the process of launching child threads from a parent thread. A child kernel performing the same or different computation can be launched using a thread from its parent kernel. Dynamic Parallelism is an extension to the CUDA and OpenCL programming models. It provides the user with constructs to implement nested thread-level parallelism. The ability to create and control workloads directly from the GPU avoids the need to transfer execution control and data between the host and the device. This reduces the overhead of invoking a GPU kernel from the host CPU. Dynamic Parallelism also offers applications the flexibility to adapt thread assignments to cores during runtime. Nested Parallelism enables the execution pattern of the application to be determined at runtime (i.e., dynamically) by the parent threads executing on the device. Additionally, since child threads can be spawned by a kernel, the GPU's hardware scheduler and load balancer are utilized dynamically to support data-driven applications.

Nested parallelism provides several performance and programmability benefits. First, recursive execution, irregular loop structures, and other complex control-flow constructs that do not conform to a single-level of task-level parallelism can be more transparently expressed. Moreover, the overhead of data transfers between kernel launches, as well as PCIe traffic, can be reduced or avoided in some cases since flow control transfers can be implemented in a single kernel. Furthermore, hierarchical algorithms can be written, where the data from a parent kernel computation is used to decide how to partition the next lower level of the computation.

3.2 Concurrent Kernel Execution

Concurrent kernel execution allows multiple kernels to execute simultaneously on the GPU. An application can launch multiple kernels containing no inherent data dependency to execute concurrently. Modern GPU architectures partition the compute resources of the GPU to enable concurrent execution. This partitioning is implemented by introducing multiple hardware queues which acquire compute resources to execute the queued kernels. Concurrent Kernel Execution can improve GPU utilization and improve the occupancy of the GPU. It can also increase cache efficiency by launching kernels which operate on the same input data.

The Hyper-Q feature was introduced by NVIDIA on their Kepler GK110 architecture devices [24]. Multiple CUDA streams get mapped to different hardware queues, which can schedule the execution of kernels on the device concurrently. Hyper-Q permits up to 32 simultaneous, hardware-managed, concurrent executions, if the kernels have no data dependency and the GPU has enough compute resources to support such execution. NVIDIA Fermi architecture GPUs map CUDA streams to a single hardware queue. This can introduce false dependencies between kernels from different streams. Such false dependencies can be avoided by using the Hyper-Q feature. Applications that were previously limited by false dependencies can see a performance increase without changing any code. The multiple streams of the applications are handled by separate hardware queues and data-independent kernels can be executed concurrently.

AMD introduced Asynchronous Compute Engines (ACE) on their GPUs as a hardware queue to schedule workloads on different compute units [15]. Work from different OpenCL command queues is mapped to different ACE units on the AMD hardware. The ACE units support interleaved execution of compute kernels on the GPU.

3.3 Memory Management

3.3.1 Shared Memory

On a GPU, shared memory is much faster than global memory. By accessing shared memory, kernels can take advantage of the lower latency provided by shared memory, and at the same time save global memory bandwidth. Moreover, shared memory can be used to avoid non-coalesced memory accesses. Applications can issue loads/stores in shared memory to reorder non-coalesced addressing.

3.3.2 Texture Memory

Texture memory is implemented on the GPU as specialized RAM that is designed for fast reads of texture data. This memory is cached in a *texture cache*, which makes a texture fetch very fast on a cache hit and costs one memory read from the device memory on a miss. The texture cache is optimized for 2D spatial locality, so the best performance can be achieved if threads in a warp read 2D texture addresses that are close together. Also, texture memory has a constant latency for streaming fetches.

Reading texture memory has four benefits over issuing reads to global or constant memory:

1. When applications exhibit 2D spatial locality, they suffer from a performance bottleneck introduced by increased global or constant memory accesses. In such cases, higher bandwidth can be achieved using texture memory fetches.
2. Address calculations are performed using dedicated hardware units.
3. Packed data may be broadcast to separate variables in a single operation.
4. Texture-memory based instructions provide an easy conversion of 8-bit and 16-bit integers to 32-bit floating-point values.

3.3.3 *Page-Locked Host Memory*

When applications use pageable memory to carry out data transfers between a host and a device, allocation of a block of *page-locked memory* is necessary. The allocation is followed by a host copy from pageable memory to a page-locked block, the data transfer, and then deallocation of the page-locked memory after the transfer. The overhead on the host for this process can be reduced when page-locked memory is directly used.

Using Page-locked host memory has three benefits:

1. Data transfers between page-locked memory and device memory can be performed concurrently with kernel execution.
2. Page-locked memory can be mapped into the address space of the device, which can avoid frequent data transfers between the host and the device.
3. Bandwidth between page-locked memory and device memory is higher on systems with a front-side bus.

3.4 Specialized Intrinsic Function

3.4.1 *Warp Shuffle Functions*

Warp shuffle functions exchange a variable between threads within a warp without the use of shared memory. All active threads simultaneously perform exchanges to transfer 4 bytes per thread per function call. The *SHFL* instruction was introduced in the CUDA programming model to implement warp shuffle.

3.4.2 *Mathematical Intrinsic Functions*

Compared to standard mathematical functions, *intrinsic functions* trade accuracy for execution speed. They execute faster since they are mapped to fewer native instructions. Intrinsic functions may also cause some differences when handling special cases such as half precision computations, divide-by-zero operations and rounding operations for transcendental functions. Therefore, standard mathematical functions are often selectively replaced by intrinsic functions in order to achieve better performance, and at the same time, generate acceptable results.

Application	**Dwarf Taxonomy**	**Domain**
Connected Component Labeling	Unstructured Grid	Object Detection
Level Set Segmentation	Structured Grid	Image Segmentation
Spectral Clustering	Spectral Method, Dense Linear	Clustering
Mean-shift Object Tracking	N-Body Method	Computer Vision
Periodic Greens Function	Dynamic programming	Electromagnetics
Infinite Impulse Response Filter	Branch and Bound	Signal Processing
Local Kernel Density Ratio	Dense Linear, Unstructured Grid	Feature Extraction
Finite-difference Time-domain	Dynamic programming	Electromagentics

Table 3: *NUPAR* applications with corresponding dwarf taxonomy and computation domain.

3.4.3 *Atomic Functions*

An *atomic function* performs a read-modify-write atomic operation on a 32-bit or 64-bit word. An atomic operation is guaranteed to be performed without interference from other threads. In other words, this address can be accessed exclusively by one thread until the operation is complete.

4. THE NUPAR BENCHMARK SUITE

The Berkeley dwarves offer benchmark application guidelines that have been used as a guiding set of principles for parallel computing benchmarks [4]. However, with the advent of GPU computing, a number of computational barriers have been overcome, enabling researchers to push the limits of applications that were previously inhibited by computing capabilities or resources. In order to stress these new parallel architecture capabilities, *NUPAR* presents eight sophisticated applications implemented using CUDA and OpenCL. Each application represents one or more dwarves. Table 3 lists the applications, along with their corresponding dwarves and general application domains.

4.1 Applications

4.1.1 *Connected Component Labeling (CCL)*

Connected Component Labeling (CCL) is a well-known labeling algorithm that is commonly used for object detection. The accuracy of the labeling process can greatly impact the fidelity of the overall object detection task. Typically, CCL performs two passes over a binary image, analyzing every pixel in an attempt to connect multiple pixels based on their position. If the current pixel is not a part of the background, then its label is determined by comparing the labels of neighboring pixels. The North and West side pixels are considered first to determine the label of a particular pixel. Once labeled, the contiguous pixels with same label are assigned the same component [36].

Our *Accelerated CCL* (ACCL) implementation uses two scanning phases [22]. The first phase scans the image in parallel in a row-wise fashion to find contiguous pixels in

Application	Programming Framework	Typical Bottleneck in an Unoptimized implementation	Optimizations Applied
CCL	CUDA	Nested loop with dependencies	Dynamic Parallelism
LSS	CUDA	Sequential execution of instances of same kernel	Dynamic Parallelism, Hyper-Q
SC	CUDA	Global Memory Bandwidth Utilization	Hyper-Q
IIR	CUDA	Synchronization Stall, Execution Dependency	Shuffle Instruction
LoKDR	CUDA	Sequential execution of data-independent kernels	Dynamic Parallelism, Hyper-Q, Local Memory
MSOT	OpenCL	Global Memory Bandwidth	Local Memory, `-cl-mad-enable`
PGF	OpenCL	Intensive Floating Point Operations	Math Intrinsics, Vector Types
FDTD	OpenCL	Intensive global memory accesses	Local Memory, Texture Memory

Table 4: Characterization of *NUPAR* applications and potential areas for optimization.

the same row that can be assigned the same label. It also creates an intermediate matrix to store the labels of each component. The second phase merges the components previously found and updates the respective labels using child threads(*dynamic parallelism*).

4.1.2 Level Set Segmentation (LSS)

Level set is an algorithmic approach commonly used in image segmentation. The goal is to partition an image into regions of interest. Using LSS, a curve is implicitly described by the level set of a multivariate surface. One significant advantage of using the level set method is that the curve can easily handle topological changes, including merges and breaks [25].

We employ the version of level set algorithm described by Shi *et al.*as the basis of our CUDA GPU implementation [29]. Points in the discretized grid are characterized in four ways: i) in L_{in} (immediately inside the curve), ii) in L_{out} (immediately outside the curve), iii) in the interior of the curve but not in L_{in}, and iv) in the exterior of the curve but not in L_{out}. Evolving the contour is simply a matter of switching points from L_{in} to L_{out} or vice versa. *Dynamic parallelism* allows us to switch points using child kernel calls, thus eliminating the need to communicate with the CPU during the image segmentation process. Additionally, multiple instances of the parent kernel can be run concurrently by utilizing NVIDIA's *Hyper-Q*, which enables more than one image to be processed at the same time.

4.1.3 Spectral Clustering (SC)

Among the many choices to perform cluster analysis, *spectral clustering* is commonly used for non-convex structures. One key advantage of spectral clustering is its ability to cluster data that is connected, but potentially sparse and unclustered within convex boundaries [20].

The spectral clustering algorithm included in *NUPAR* is a matrix-based implementation which is well-suited for GPUs. We define 4 steps present in the algorithm based on the Ng-Jordan-Weiss approach [20]: (1) form the affinity matrix, (2) apply Laplacian normalization, (3) run an eigen solver and (4) cluster using k-means. Using multiple kernels, we can leverage the latest features offered by the CUDA framework such as: *Hyper-Q and pinned memory.* Our parallel approach uses a tile-based approach and *Hyper-Q* allows execution of concurrent kernels over different tiles. *Dynamic parallelism* is used to compute the nearest cluster during the k-means step. *Pinned memory* leverages the higher bandwidth between host memory and the device memory.

4.1.4 Mean-shift Object Tracking (MSOT)

Mean-shift is an algorithm for tracking non-rigid objects in a sequence of images. The mean-shift algorithm uses the color histogram of the target object in the current frame, and iteratively searches the neighborhood in the next frame to find the location whose color histogram is closest to the target. The Bhattacharyya coefficient is used to measure the distance between two histograms [9].

We implement the Mean-shift algorithm using multiple levels of granularity with OpenCL. At a coarse-grained level, the program tracks multiple objects at the same time. The program also calculates the histogram for the neighbors of each object to reduce the total number of iterations and therefore the execution time. Working at a fine-grained level, the histogram calculation is distributed to the work-items in a work-group. Each work-item calculates a portion of the histogram – we use a reduction method to compute the overall histogram. We also use a lookup table to reduce the memory size and shared memory usage. With our parallel approach, we can modify the work-group/work-item ratio in order to stress a specific GPU architecture.

4.1.5 Periodic Green's Function (PGF)

The *Periodic Green's Function* is a discrete implementation of the continuous function used in the integral equation (IE) to solve computational electromagnetics (CEM) problems by application of the Method of Moments (MoM) [30]. In its standard representation, the PGF involves a slowly converging series of free space Green's Function. When applying a windowed summation method, we are able to compute the PGF several million times for different points in the lattice [6]. This accelerated function evaluation fits into a larger array integral equation designed to parallelize infinitely periodic electromagnetic problems. When the problem of interest involves infinite periodic structures, the PGF provides a fast and efficient method to solve CEM problems with a windowed summation method, requiring the evaluation of the PGF on the order of several million times. Our OpenCL GPU code achieves the best performance when the number of PGF evaluations is large, as we assign each work-item a unique call to the PGF.

When considering alternative data types, we chose `double2` for our complex output type to improve the global memory store efficiency and improve memory transfers from the device to the host. In an effort to reduce the computational cost of the large number of transcendental functions and multiply-add calculations, we replace these functions with their equivalent math intrinsic calls and use the "cl-finite-math-only" compiler flag during clBuildProgram.

4.1.6 Infinite Impulse Response Filter (IIR)

Finite Impulse Response (FIR) and *Infinite Impulse Response* (IIR) are used in signal processing applications such as speech and audio processing. IIR is preferred to FIR if some phase distortion is either tolerable or unimportant.

Application	Baseline Implementation	Application Input	Baseline GPU
CCL	CUDA with less workload on child kernels	5 Images (512 x 512)	NVIDIA K40
LSS	CUDA with fewer threads per child kernel	Over 256 (512 x 512) Images	NVIDIA K40
SC	CUDA with serial kernel call(non-Hyper-Q)	Tile Basis of 900 elements (Only first two kernels)	NVIDIA K40
IIR	CUDA with Shared Memory Use (without *SHFL* instruction)	Input 1024 (floats) 128 Channels, 256 parallel biquad filters/channel	NVIDIA K40
LoKDR	CUDA with serial kernel calls (non-Hyper-Q) and no nested parallelism	Input data set with 7129 features and 99 samples	NVIDIA K40
MSOT	OpenCL without shared memory and cl-mad-enable	10 objects tracked over 120 frames	AMD Radeon 7970
PGF	OpenCL without cl-finite-math-only and fma, hypot	Number of PGF Evaluations = 3 million, FFT Sample Rates = (120,120,120)	AMD Radeon 7970
FDTD	Naive OpenCL without use of local and Texture memory	(nx, ny, nz) = (240, 80, 80), (dx, dy, dz) = (0.005, 0.005, 0.005)	AMD Radeon 7970

Table 5: Baseline configuration for each *NUPAR* application and associated input datasets.

Due to its nature, IIR performs a smaller number of calculations per time step than FIR.

Our parallel implementation of IIR [26] starts by decomposing the transfer function into parallel second-order IIR sub-filters. Each sub-filter reduces the waiting period for the next output calculation. A multi-channel high order IIR should improve IIR filtering efficiency on a GPU [28].

We have implemented a parallel IIR using the CUDA framework, leveraging the *shuffle (SHFL)* instruction on the Kepler architecture to achieve a fast summation to produce the output signal. Each channel's coefficients are cached in constant memory along with the input signal. Filtering multiple channels works in a block-wise fashion (i.e., the number of channels is equal to the number of blocks launched on the GPU). Our grid configuration attempts to stress GPU occupancy by increasing the number of channels.

4.1.7 Local Kernel Density Ratio (LoKDR) for Feature Selection

Selecting the best set of features is an important step in all machine learning tasks. The focus is to determine and choose features relevant for classification and regression of the given data. Our next application presents a non-parametric evaluation criterion for filter-based feature selection to enhance outlier detection [5]. The method identifies the subset of features that represents the inherent characteristics of the normal dataset, while also identifying features to filter out outliers.

Our CUDA application utilizes GPUs for computation of the *K-nearest neighbors* in a dataset. These kernels have to be launched for each feature and each set of features to determine outliers. Data parallelism offered by the computation is nicely exploited using the GPU. Computation includes kernels to compute the distance between pairs of points in the data set and includes sorting of the k-nearest neighbors. We utilize *dynamic parallelism* and develop a pipelined approach for outlier detection for each feature. *Hyper-Q* is used to facilitate concurrent execution of these pipelined kernels. Computations launched for different features are also executed concurrently.

4.1.8 Finite-Difference Time-Domain (FDTD)

The *Finite-Difference Time-Domain* (FDTD) method is a widely-used computational method for solving Maxwell's equations in many electromagnetics problems. FDTD is a grid-based marching-in-time algorithm that calculates the electric and magnetic fields over every cell in a computational domain at each time step [18].

When using FDTD, the entire computational domain needs to be divided into a number of Yee cells whose size must be sufficiently small to resolve smaller electromagnetic wavelength and smaller geometrical features in the model. A medium-sized computational domain can often lead to days or even weeks of solution time. Thus, we employ GPUs as a highly multi-threaded data-parallel processor to accelerate FDTD simulation. In our implementation of FDTD, up to 35 kernels are launched for the computation based on the size of the input.

For each iteration, we advance the time by Δt, which is a configurable unit for the benchmark. FDTD computes the H(magnetic) and E(electric) fields for each cell. Then, the computation advances time by Δt and begins the next iteration. A leap-frog integration scheme is used to compute the H and E values of each cell. The value at time t is dependent on the previous value at time step $t-\Delta t$, and thus sequential iterations cannot be run in parallel. However, the computation of each cell at a specified time t is independent of other cells, and so can be run in parallel.

5. EVALUATION METHODOLOGY

Next, we describe the platforms used for evaluating the performance of the *NUPAR* applications, implemented in either CUDA or OpenCL. The baseline used for evaluating each application is also described in this section.

5.1 Evaluation Platforms

The NVIDIA K40 and the AMD Radeon HD 7970 GPUs are used to evaluate the *NUPAR* applications developed using CUDA and OpenCL, respectively. These platforms are comparable in terms of clock rate, number of cores, peak bandwidth and FLOPS. Table 6 compares the specifications

	NVIDIA	AMD
Device Name	K40	HD7970
GPU Architecture	Kepler	Southern Islands
Peak Single Precision FLOPS (Gflops)	4291	3789
Peak Bandwidth (GB/s)	288	264
Streaming Cores	2880	2048
Clock Rate (MHz)	876	925
Global Memory (GB)	12	3
L2 Cache (KB)	1536	768
Constant Memory (KB)	64	128
Shared Memory Per Block (KB)	48	64
Warp/Wavefront Size	32	64

Table 6: Architectural specifications of GPU platforms used for evaluation.

of these two GPU platforms. The CUDA workloads are profiled using NVIDIA's Visual Profiler. AMD's CodeXL profiler is used for profiling OpenCL workloads. The applications supporting CUDA were developed for the CUDA-6 version of the programming framework. The OpenCL applications were developed for both OpenCL 1.2 and OpenCL 2.0 versions. The OpenCL 2.0 applications utilize modern features described in Section 3. Due to the unavailability of platforms supporting OpenCL 2.0 during the time of development of this paper, the evaluation of applications on OpenCL 2.0 has not been performed yet. The current evaluations of the OpenCL applications are done with the OpenCL 1.2 version of the programming framework.

5.2 Baselines

The *NUPAR* suite focuses on exercising new GPU architectural and programming features. Each application in the *NUPAR* benchmark suite has been optimized using one or more features to tackle an inherent bottleneck present in the unoptimized implementation. To evaluate the *NUPAR* workloads, each application is compared with an unoptimized implementation as its baseline. The SC workload uses Hyper-Q as an optimization feature. This workload is compared with the same CUDA implementation with serial kernel calls instead of using Hyper-Q. The CCL and LSS workloads are also compared with a CUDA version that places less workload on child kernels and uses fewer threads per child kernel, respectively. The baseline for MSOT and FDTD use their original OpenCL implementations, their optimized runs use shared or texture memory and the `cl-mad-enable` optimization. Table 5 summarizes the baselines used for evaluating each application.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the *NUPAR* applications on the platforms described in Section 5. We also investigate the performance benefits obtained when applying the optimizations described in Section 3.

6.1 Computational Characteristics of NUPAR Applications

Each of the *NUPAR* applications stresses different architectural features on the GPU or in the updated programming models. We characterize the behavior of the applications based on seven factors: *i) Global memory bandwidth utilization, ii) Occupancy, iii) Register Utilization, iv) Local/Shared memory usage, v) Control Flow/Divergence, vi) Cache utilization*, and *vii) ALU usage.*

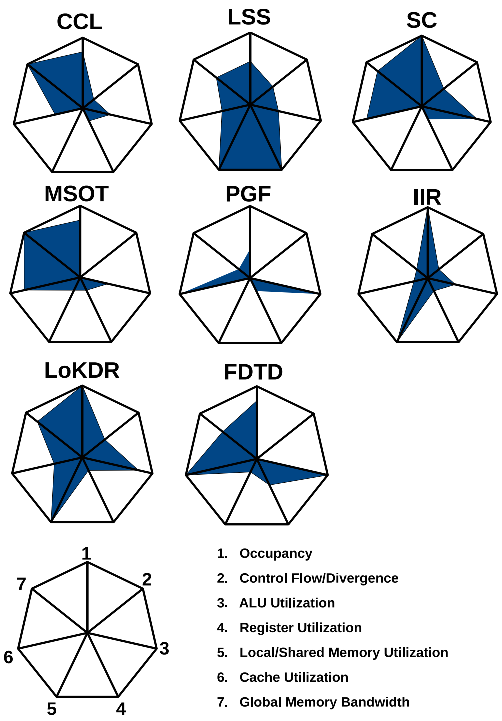

Figure 1: Kiviat-charts describing computational characteristics of *NUPAR* applications.

Figure 1 presents a Kiviat-chart to clearly characterize the different architectural features exercised by each *NUPAR* application. Each axis of the Kiviat diagram denotes a different feature. The values for each architectural feature are obtained using the profiling tools described in Section 5. The values show the percent utilization for each architectural feature. The applications from the *NUPAR* suite exhibit diversity in their characteristics, as seen in Figure 1. For example, the *CCL, SC, MSOT, and LoKDR* applications show heavy utilization of the global memory bandwidth, whereas *PGF and FDTD* utilize the cache on the GPU. Local/shared memory performance can be evaluated using the *LSS, IIR and LoKDR* applications.

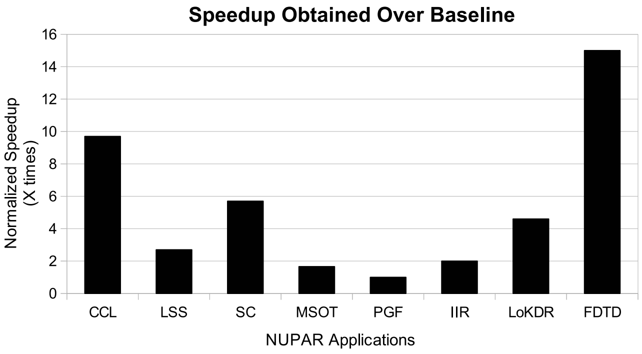

Figure 2: Normalized speedup of *NUPAR* applications over their corresponding baseline.

6.2 Speedup Analysis

We evaluate the speedup obtained by the *NUPAR* applications after applying the optimizations required to sensitize the new architectural features that were described in Section 3. The baseline for each application is described in Table 5.

The speedup obtained by the benchmark applications is shown in Figure 2. The average speedup observed for all applications is 5.1x. The *FDTD* sees the greatest speedup by using local memory and texture memory available on the GPU. Applications such as *MSOT* and *PGF* benefit by using optimized multiply-add instructions for integer and floating point operations. Dynamic parallelism is used in *CCL* to cache the data for sub-kernels on the device without transferring intermediate results back to the host and then back to the device. *CCL* achieves an overall speedup of 9.7x by launching multiple image processing kernels simultaneously. *LoKDR* also sees a 4.3x speedup using NVIDIA's Hyper-Q, which allows for concurrent execution of data-independent kernels. Similarly, a speedup of 5.9x is seen for *SC* and speedup of 2.3x is seen for *LSS* when using Hyper-Q and dynamic parallelism. *IIR* utilizes the *SHFL* intrinsic for performing inter-thread memory operations and sees a speedup of 2.2x.

Figure 3 shows the pre-optimization (baseline) and post-optimization resource utilization mix of the *NUPAR* applications. *LSS* and *LoKDR* experience an increase of 19% in occupancy, improvements in cache utilization by 7%, and a rise in ALU utilization by 9% on average after applying the optimizations. For *SC*, the occupancy increases from 74% to 92% by using the Hyper-Q feature. As observed, an increase in occupancy is tied to a more efficient use of the cores through concurrent kernel execution using Hyper-Q. For the *IIR* benchmark, the use of the *SHFL* instruction increases the number of blocks that can be allocated on each stream processor. This leads to an 18% improvement in occupancy, and an increase in cache utilization from 21% to 26%, and an increase in ALU utilization from 22% to 28% for the *IIR* application.

6.3 Performance of Nested Parallelism

Nested parallelism is described in Section 3 and is referred to as *Dynamic Parallelism* by the CUDA and OpenCL programming frameworks. Many classes of algorithms (e.g., *CCL* and *LoKDR*) can potentially benefit from *Dynamic Parallelism. CCL* requires thread-level parallelism to reduce the overhead of updating labels of multiple components that belong to the same object during its detection of connected components. Similarly, *LoKDR* utilizes dynamic parallelism to detect outliers by calculating the distance between the nearest neighbors. The overhead of launching kernels to perform the distance calculation is reduced through dynamic parallelism.

We evaluate the speedup obtained using nested parallelism for the *CCL*, *LSS* and *LoKDR* applications. Figure 4 shows the speedup obtained for these applications when varying the number of threads per child kernel. We observe that we achieve more speedup as we increase the number of threads per child kernel. This can be attributed to the increase in overall occupancy of the GPU. Increasing the number of threads for the child kernel reduces the effective number of child kernel launches. This avoids the overhead of launching multiple child kernels on the GPU. An increased number of threads per child kernel results in higher ALU utilization for these three applications, as illustrated in Figure 3b. The peak speedup reported by *CCL*, *LSS* and *LoKDR* is 13.9X, 5.8X and 6.2X, respectively. The *CCL*, *LSS* and *LoKDR* applications can be used to evaluate how the number of threads per child kernel impacts the performance when using dynamic parallelism.

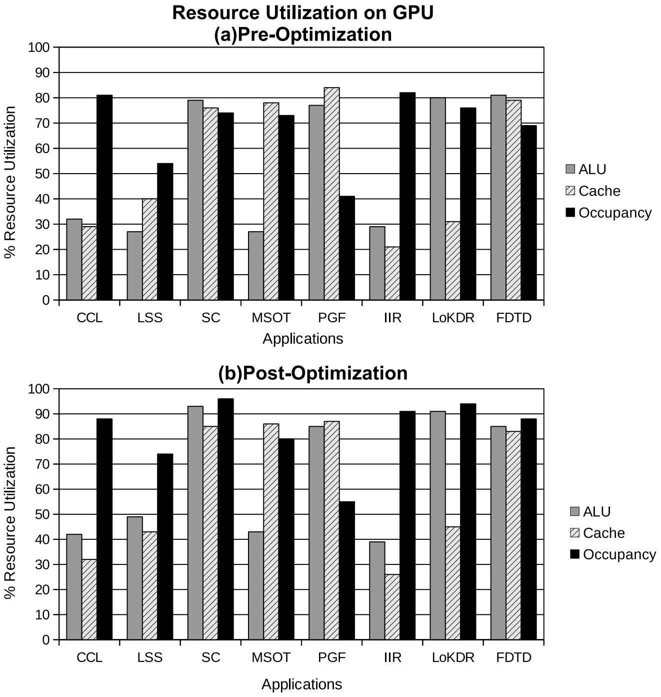

Figure 3: Resource utilization of the *NUPAR* applications, (a) Pre-Optimization and (b) Post-Optimization.

The communication between the threads of the parent kernel and child kernel can be carried out only using global memory for dynamic parallelism. Applications such as *CCL, LSS and LoKDR* impose a heavy communication penalty due to communication between the parent and child threads. Figure 5 shows the throughput obtained for accessing data between parent and child kernels while varying the number of threads per child kernel. Figure 5 shows that our CUDA implementation for *CCL*, *LSS* and *LoKDR* achieves higher global memory throughput as we increase the work on the child kernel. The average global memory throughput achieved by *CCL*, *LSS* and *LoKDR* is 14.6 Gb/s, 16.1 Gb/s and 15.3 Gb/s, respectively. The number of coalesced global memory accesses increases as we increase the number of threads for a kernel. The efficient utilization of the global memory load/store unit results in a clear improvement in global memory throughput. These benchmarks can be used to judge the global memory throughput of the GPU.

6.4 Utilizing Concurrent Kernel Execution

Concurrent kernel execution is an important feature supported by modern GPUs, as described in Section 3. NVIDIA GPUs utilize the new Hyper-Q mechanism and AMD GPUs use the ACE (Asynchronous Compute Engine) units to man-

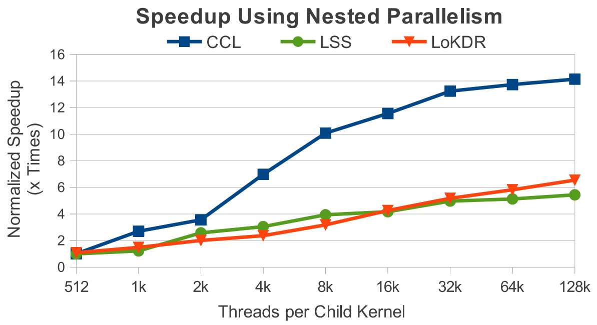

Figure 4: Speedup obtained using nested parallelism for *CCL*, *LSS* and *LoKDR*. Speedup is reported while varying the number of threads per child kernel.

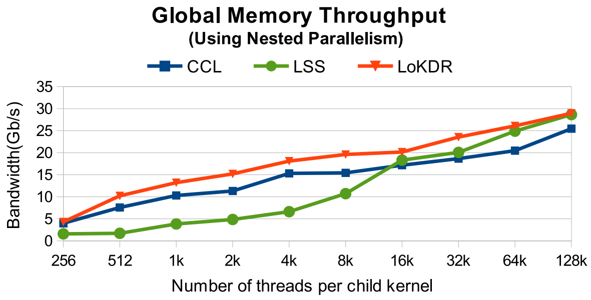

Figure 5: Evaluation of global memory throughput for parent-child kernel communication for *CCL*, *LSS* and *LoKDR* using nested parallelism.

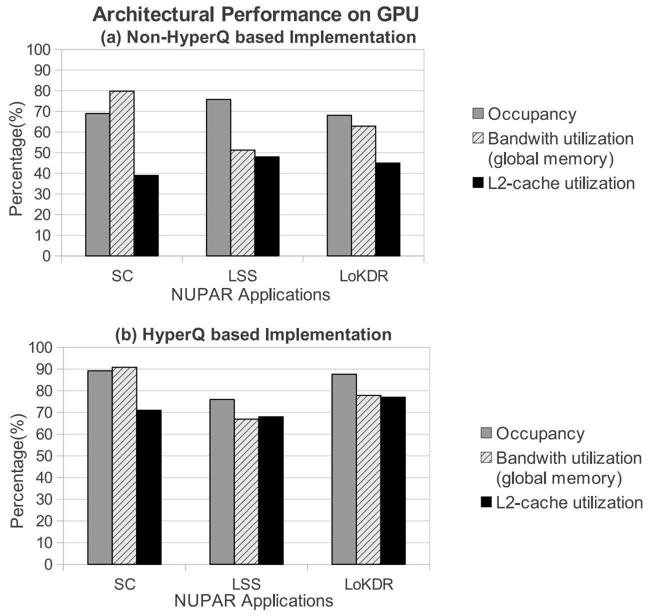

Figure 6: Comparison of architectural performance of NUPAR applications on GPUs for (a) Non-Hyper-Q based implementation and (b) Hyper-Q based implementation

age concurrent kernel execution. We explore the performance of this class of features using three different applications from the *NUPAR* suite: *SC, LSS, LoKDR.*

The occupancy improvement of any implementation that uses multiple streams depends on the resources utilized by each stream. As seen in Figures 6a and 6b, *LSS* and *LoKDR* do not find any improvement in occupancy. This is because the kernels executed in each stream have already saturated the available resources. *SC* exploits a new level of parallelism exposed by Hyper-Q, since each stream occupies less than 70% of the GPU when executed sequentially.

Concurrent execution of kernels on the GPU using Hyper-Q also results in improved L2 cache utilization. We can see in Figure 6 that applications such as *LoKDR* and *LSS* see an improvement of 32% and 20% in cache utilization, respectively. The overall cache utilization of all the kernels launched by the application is recorded using the profiler tools. Hyper-Q facilitates concurrent execution of kernels which operate on same input data (e.g., *LoKDR*), which results in an overall improvement in cache utilization.

We have also studied the global memory bandwidth utilization for all three applications. Figure 6 shows the positive impact of Hyper-Q on the memory efficiency. Hyper-Q helps concurrent kernels keep the load/store units on the GPU busy. This improves global memory bandwidth utilization of the GPU. The *LoKDR* and *LSS* applications experience an increase greater than 15% in terms of global memory bandwidth utilization. The *SC* application already has high global memory bandwidth utilization and does not show a significant improvement in bandwidth utilization when using Hyper-Q. These three applications can also be used to test queuing mechanisms on the GPU by using concurrent execution of kernels.

6.5 Specialized Intrinsics and Instructions

Programming frameworks such as CUDA and OpenCL from time to time introduce specialized intrinsic functions and compiler optimizations for improving performance of complex computations. Intrinsic functions provide the user with the ability to perform complex math calculations efficiently, including execution of transcendental operations, square-root operations and atomic operations. Different compiler optimizations can leverage new GPU hardware features and carry out operations using special instructions.

For cases where a potential reduction in accuracy is tolerable, fast math operation intrinsics can be turned on during GPU code generation. Use of these intrinsics shrinks the size of the executable binaries by reducing the number of instructions executed on the GPU. The *NUPAR* applications, such as *MSOT* and *PGF*, use the fast-math intrinsics for compile-time optimizations.

MSOT uses the `-cl-mad-enable` optimization option, which leverages special instructions for multiply-add operations. The impact on a range of performance factors is shown in Table 7. Using the `-cl-mad-enable` switch reduces vector register usage per work-item by 87%. This results in an increase in the number of active waves, which enhances the kernel occupancy by 60%. The improvement in such factors is responsible for the 1.9x speedup obtained by *MSOT*, as seen in Figure 2. Besides the computational speedup, the occupancy ratio also affects the kernel execution time and can be improved by reducing the amount of

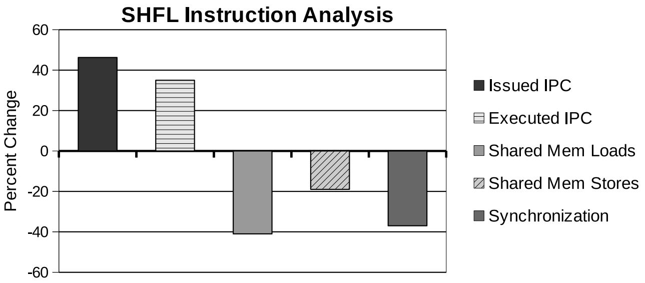

Figure 7: Percentage change in performance metrics due to the use of *SHFL* instruction in *IIR* application over the baseline.

shared memory usage. NVIDIA's Kepler architecture provides new *SHFL* (shuffle) instructions as an efficient solution for fast intra-warp communications. The use of the *SHFL* reduces shared memory usage. Though inter-warp communication still requires some shared memory, the total amount of shared memory used could ideally be lowered by 32 times [1]. As a result, the number of instructions and barriers for *SHFL* are less than the number of instructions using shared memory.

The *IIR* consists of multiple parallel biquad filters. The output signal is produced by summing up the independent biquads results for each time step. The Parallel reduction used in *IIR* is an ideal fit for the shuffle instructions. It is required for summation among the threads where each thread functions as a biquad filter. We observe a 2.1x speedup in kernel execution time when launching 128 channels for a parallel IIR program. We use the *nvvp* profiler to extract the related performance counters for the execution of *IIR*. As shown in the Figure 7 (which includes the selected performance counters), the *SHFL* instruction increases the Instructions Per Cycle (IPC) by 35%. The number of synchronizations is reduced by 38% and the shared memory usage (load/store) drops by 30% on an average.

Performance Counters	Un-Optimized	Optimized
Vector Registers per Work-item	167	22
Number of waves limited by vector registers	40	4
Number of active waves	4	28
Occupancy	10.00%	70.00%

Table 7: Comparison of performance counters for un-optimized and optimized versions of MSOT.

Floating point performance is an important metric to evaluate overall GPU performance. Double precision optimizations are included in the vendor-specific implementations of many FFT and linear algebra routines. We include the periodic Green's function which requires millions of floating point calculations per iteration for evaluating the double precision performance of the GPU.

We use the fused-multiply-add (FMA) optimization to evaluate floating point performance of the GPU using the *PGF* application. Figure 8 shows the execution time of the *PGF* application when varying the number of *PGF* evaluations. The analysis is done for *Single Precision, Double Precision and Double2* variables, with and without FMA support. As observed in Figure 8, the computations using single precision variables and using FMA yield the best execution performance. Double precision operations use two single precision registers and operations are performed by specialized double precision units on modern GPUs. An average slowdown of 9.3% is observed due to the use of double precision over single precision. FMA optimizations force the hardware to utilize instructions for the specialized double precision units on the GPU. The double precision performance with FMA shows an average speedup of 3.9% over the regular double precision execution.

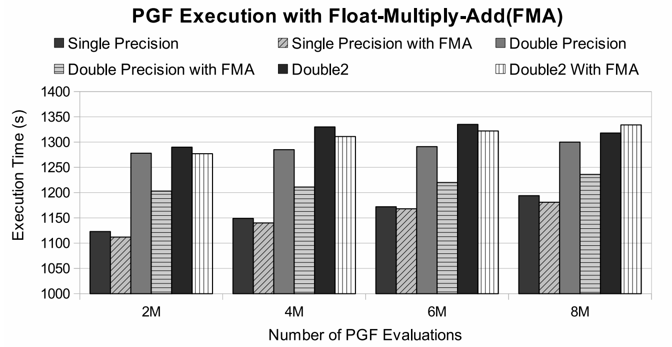

Figure 8: Kernel time comparison of PGF for 2 to 8 million PGF evaluations using different precision types to represent the complex output of the PGF.

We also tested the performance of the *PGF* with the `double2` complex output data type. As seen in Figure 8, the `double2` evaluation shows a minor loss in performance as compared to the real and imaginary `double` kernel code. The differences between the two versions are exaggerated due to the scaling of the Y-axis in Figure 8.

7. DISCUSSION

NUPAR covers eight applications across a wide spectrum of domains. The features described in Section 3, such as Nested Parallelism, Concurrent Kernel Execution, Shared memory, and Texture memory are available on both the OpenCL and CUDA frameworks. In *NUPAR*, we use features implemented in CUDA 6. The same features are also available on OpenCL 2.0, and can be implemented when hardware supporting OpenCL 2.0 is available [17]. Table 8 demonstrates the features available in OpenCL 2.0 and CUDA 6. Nested Parallelism is referred to as *Dynamic Parallelism* in both CUDA and OpenCL terminologies. Concurrent Kernel Execution is referred to as *Hyper-Q* in CUDA. The OpenCL 2.0 specification also incorporates Shared Virtual Memory and Pipe-based communication channels as advanced features. Shared Virtual Memory is known as unified virtual addressing in CUDA terminology and was introduced in CUDA-4. The advanced constructs such as *SHFL* are only supported on CUDA architectures developed by NVIDIA.

The new architectures developed using the HSA (Heterogeneous System Architecture) standard will provide advanced features such the shuffle intrinsic [27]. HSA also supports other new features, such as the Architected Query Language (AQL), a simple job queuing mechanism which handles the memory transfers and kernel execution. Another important feature for Inter-Kernel Communication known as *Pipes* is supported on HSA and OpenCL 2.0, but is not

	CUDA 6	OpenCL 2.0
Nested Parallelism	✓	✓
UVA	✓	✓
Concurrent Kernel Execution	✓	-
Inter-Kernel Communication	-	✓
Atomics	✓	✓

Table 8: Comparison of Programming Features Between CUDA 6 and OpenCL 2.0

yet part of the CUDA standard. The present HSA standards for GPUs and accelerators are supported by many vendors such as AMD, ARM, Qualcomm, Samsung, and Imagination. Many of the features introduced in HSA and OpenCL 2.0 will be supported on hardware provided by all of these vendors. The *NUPAR* suite can be used for evaluating the performance of this new class of GPUs.

8. RELATED WORK

Several benchmark suites have been developed in the past for evaluating the performance and architectural characteristics of GPU systems. Rodinia [7] and Parboil [31] benchmark suites target GPUs and multi-core CPUs using CUDA. However, their scope is limited to traditional GPU architectures and do not exercise the newer hardware features of modern GPUs and other accelerators. The *NUPAR* suite fulfills this requirement of GPU benchmarks by targeting modern heterogeneous systems. SHOC is a scalable benchmark suite implemented in CUDA and OpenCL to measure the performance and stability of both NVIDIA and AMD platforms [10]. Valar also targets both AMD and NVIDIA devices, while focusing on evaluating the interaction between host and device in heterogeneous systems [16]. We developed the *NUPAR* suite for CUDA and OpenCL to provide the user with programming flexibility. Volkov and Demmel have benchmarked linear algebra applications using CUDA on NVIDIA GPUs [34]. *NUPAR*, however, covers a larger spectrum of applications. CUDA-NP proposes a compiler-level solution to leverage nested parallelism for GPGPU applications [35]. Y. Liang et al. demonstrate a performance improvement over Hyper-Q using their technique which allows spatial and temporal multitasking on GPUs [8]. But they do not provide researchers with a set of applications to evaluate these features on the GPUs.

Another common way to benchmark GPUs includes measurement of frames per second (FPS) achieved by computationally demanding games such as *Crysis* [11]. Also, proprietary GPU benchmark softwares such as 3DMark are designed to determine the performance of GPUs using DirectX [2]. Other parallel benchmark suites for CPUs that have been developed include MediaBench [14] for multimedia applications, BioParallel [12] for biomedical applications, and MineBench [19] for data mining applications. These benchmark suites target a specific application domain and do not provide a diverse range of workloads. Lonestar is an attempt to extract amorphous data-parallelism from graph-based real world applications [13]. *NUPAR* is distinct from these works as it primarily targets modern GPUs using OpenCL and CUDA and provides a set of diverse applications to highlight changes in the evolving GPU architectures.

9. CONCLUSION AND FUTURE WORK

In this paper, we have introduced the *NUPAR* benchmark suite designed to provide a rich set of parallel programs to study the performance of emerging architectural features and programming constructs targeting modern heterogeneous platforms. The applications help the user judge the performance of the new class of GPUs and accelerators with features such as nested parallelism, concurrent kernel execution and advanced computational and memory instructions. We provide eight publicly available implementations of real-world applications belonging to different scientific domains. The applications are developed using both CUDA and OpenCL programming frameworks. The paper characterizes the different applications according to the architectural features stressed by such applications. We also highlight the performance obtained by the use of different architectural optimizations as described in the paper.

We plan to support the *NUPAR* applications on different OpenCL-compatible platforms including FPGA platforms and embedded SoCs from vendors such as Qualcomm and NVIDIA. We would like to extend the *NUPAR* suite to include applications developed for graphics and for interoperable compute-graphics. We are planning to add additional applications which utilize the unified virtual memory model introduced in CUDA 6 and OpenCL 2.0. The benchmark suite is available for download *https://code.google.com/p/nupar-bench/*.

Acknowledgments

This work was supported in part by a NSF CISE award CSR-1319501. We would like to thank the HSA foundation for their gift to support this work. We would also thank AMD and Nvidia for providing hardware to conduct this work. We thank Fatemeh Azmandian for her help with the *LoKDR* application development.

10. REFERENCES

[1] CUDA C Programming Guide. *NVIDIA Corporation, Feb*, 2014.

[2] 3DMark. http://www.futuremark.com/benchmarks.

[3] AMD. Accelerated Parallel Processing: OpenCL programming guide. *URL http://developer. amd. com/sdks/AMDAPPSDK/documentation*, 2011.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. 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.

[5] F. Azmandian, A. Yilmazer, J. G. Dy, J. A. Aslam, and D. R. Kaeli. Gpu-accelerated feature selection for outlier detection using the local kernel density ratio. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, pages 51–60. IEEE Computer Society, 2012.

[6] O. P. Bruno, S. P. Shipman, C. Turc, and V. Stephanos. Efficient Evaluation of Doubly Periodic Green Functions in 3D Scattering, Including Wood Anomaly Frequencies. *Arxiv*, 1307.1176:80–110, 2013.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[8] D. Chen, H. P. Huynh, R. S. M. Goh, and K. Rupnow. Efficient gpu spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, page 1, 2014.

[9] D. Comaniciu, V. Ramesh, and P. Meer. Real-time tracking of non-rigid objects using mean shift. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*, volume 2, pages 142–149, 2000.

[10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. 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.

[11] C. Frankfurt. Crysis. http://www.crysis.com, 2007.

[12] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (llc) performance of data mining workloads on a cmp-a case study of parallel bioinformatics workloads. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 88–98. IEEE, 2006.

[13] M. Kulkarni, M. Burtscher, C. Casçaval, and K. 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.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[15] M. Mantor and M. Houston. AMD Graphics Core Next. In *AMD Fusion Developer Summit*, 2011.

[16] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli. Valar: A benchmark suite to study the dynamic behavior of heterogeneous systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 54–65. ACM, 2013.

[17] A. Munshi. The OpenCL Specification 2.0. *Khronos OpenCL Working Group*, 2014.

[18] T. Namiki. A new fdtd algorithm based on alternating-direction implicit method. *Microwave Theory and Techniques, IEEE Transactions on*, 47(10):2003–2007, 1999.

[19] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188. IEEE, 2006.

[20] A. Y. Ng, M. I. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2:849–856, 2002.

[21] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[22] F. Nina-Paravecino and D. Kaeli. Accelerated connected component labeling using cuda framework. In *Computer Vision and Graphics (ICCVG), 2014 International Conference on*, 2014.

[23] NVIDIA. Visual Profiler, 2011.

[24] NVIDIA. NVIDIA's Next Generation CUDA Computer Architecture Kepler GK110. 2012.

[25] S. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed: algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[26] B. Porat. *A course in digital signal processing*, volume 1. Wiley New York, 1997.

[27] P. Rogers. Heterogeneous system architecture overview. In *Hot Chips*, 2013.

[28] J. Schmidt. A Flexible IIR Filtering Implementation for Audio Processing. *Technicolor Research & Innovation, GTC*, 2014.

[29] Y. Shi and W. C. Karl. A fast implementation of the level set method without solving partial differential equations. *Boston University, Department of Electrical and Computer Engineering*, 2005.

[30] S. Singh, W. F. Richards, J. R. Zinecker, and D. R. Wilton. Accelerating the convergence of series representing the free space periodic green's function. *Antennas and Propagation, IEEE Transactions on*, 38(12):1958–1962, 1990.

[31] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[32] J. A. Stratton, S. S. Stone, and W. H. Wen-mei. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer, 2008.

[33] W. Sun and R. Ricci. Augmenting Operating Systems With the GPU. 2011.

[34] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008.

[35] Y. Yang and H. Zhou. Cuda-np: realizing nested thread-level parallelism in gpgpu applications. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 93–106. ACM, 2014.

[36] H. Zhao, Y. Fan, T. Zhang, and H. Sang. Stripe-based connected components labelling. *Electronics Letters*, 46:1434–1436, October 2010.