Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing

John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid,
Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, Wen-mei W. Hwu
{stratton, cirodrig, sung10, obeid1, lchang20, assari1, gengliu2, w-hwu}@illinois.edu

IMPACT Technical Report
IMPACT-12-01
University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
March 2, 2012
Revision: March 19, 2012

Abstract

The Parboil benchmarks are a set of throughput computing applications useful for studying the performance of throughput computing architecture and compilers. The name comes from the culinary term for a partial cooking process, which represents our belief that useful throughput computing benchmarks must be "cooked", or preselected to implement a scalable algorithm with fine-grained parallel tasks. But useful benchmarks for this field cannot be "fully cooked", because the architectures and programming models and supporting tools are evolving rapidly enough that static benchmark codes will lose relevance very quickly.

We have collected benchmarks from throughput computing application researchers in many different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy. Each benchmark includes several implementations. Some implementations we provide as readable base implementations from which new optimization efforts can begin, and others as examples of the current state-of-the-art targeting specific CPU and GPU architectures. As we continue to optimize these benchmarks for new and existing architectures ourselves, we will also gladly accept new implementations and benchmark contributions from developers to recognize those at the frontier of performance optimization on each architecture.

Finally, by including versions of varying levels of optimization of the same fundamental algorithm, the benchmarks present opportunities to demonstrate tools and architectures that help programmers get the most out of their parallel hardware. Less optimized versions are presented as challenges to the compiler and architecture research communities: to develop the technology that automatically raises the performance of simpler implementations to the performance level of sophisticated programmer-optimized implementations, or demonstrate any other performance or programmability improvements. We hope that these benchmarks will facilitate effective demonstrations of such technology.

I. Introduction

Benchmarks are an incredibly important part of computer architecture research. They allow a research community to focus on a shared codebase, with individual researchers more easily able to understand another's results. If a benchmark or set of benchmarks is generally accepted, its use can dispel concerns of bias that often arise when a researcher creates their own benchmarks to generate their published data. The specific goals of existing benchmark suites are as varied as the suites and research communities themselves, but all focus on defining a limited set of codes representative of some larger set that is worth studying. Most benchmark suites also explicitly or implicitly define a category of computer system that the benchmarks are most useful for studying. As computer systems change over time, older benchmarks become less useful, motivating the creation of new benchmarks applicable to the current systems and applications of interest.

Throughput computing, especially targeting a heterogeneous CPU+Accelerator system architecture, has been steadily growing and evolving as an application and architecture area of great interest. The most widely studied variant thus far has been the CPU+GPU accelerator system architecture, typically using the OpenCL and NVIDIA CUDA languages. As the properties and challenges of programming GPU architectures were understood more and more, architecture and compiler research in the area of throughput computing has also grown significantly. Yet throughput computing research remains difficult to evaluate as there have been few consistent, useful benchmarks to use as test cases. Many authors created their own benchmarks out of necessity, but may not explicitly publish the code used to create their published works, making independent replication or validation of results extremely difficult.

To address this situation, an appropriate benchmark suite should:

- Represent the emerging scientific and commercial application uses of throughput computing.
- Represent application algorithms exhibiting low algorithmic complexity.
- Scale to very wide parallel architectures, anticipating the potential for exponential growth in machine width and choosing algorithms that will remain applicable for several years.
- Provide application implementations and a benchmarking infrastructure that will support the work of various research interests, at minimum encompassing architecture, microarchitecture, compiler, language, and programming environment research.

Taking these directives as our goals, we have developed the new Parboil benchmark suite. The Parboil benchmarks are a set of accelerated, heterogeneous applications emphasizing throughput-oriented computing. The benchmark suite includes iterative methods, dynamic task kernels, dense array operations, and data-dependant memory access patterns. The diversity of the kernels within the Parboil benchmarks exercises memory bandwidth, floating-point throughput, latency tolerance, and even cache effectiveness, with applications drawn from astronomy, biomolecular simulation, fluid dynamics, image processing, astronomy, and dense and sparse linear algebra. Each benchmark is based on a specific algorithm that is scalable and applicable for the datasets included with the benchmarks. Furthermore, each benchmark includes multiple implementations of that algorithm to support different benchmarking comparisons or scenarios.

First, we provide a baseline accelerated version of every benchmark, which embodies a reasonable throughput-oriented implementation. The baseline accelerated version represents reasonable decisions about how to parallelize and accelerate the algorithm computationally, but makes little optimization effort. Such versions are often constrained by memory bandwidth, because they do not necessarily have good coalescing behavior or other typical optimizations. We believe these versions are similar to what the average programmer would write as a first functionally-correct effort, and possibly the deployed version if further optimization is more costly than the programmer can afford.

Second, we provide example architecture-optimized versions, on which significantly more programmer effort was typically invested to avoid performance pitfalls or architecture shortcomings for a specific device, representing the state of the art for that application on that architecture to the best of our knowledge. The initial architecture-optimized examples highlight the gap between the performance possible and the performance achieved by the baseline version. The initial optimized versions also demonstrate for other researchers some of the potential optimizations that have worked well for each application, should they choose to develop a new implementation advancing the state of the art.

Which leads us to the third kind of implementation included in each benchmark. Anticipating that researchers may want to reimplement the application algorithms using a different language or framework, we provide a simple,

readable, C or C++ implementation to help facilitate the reimplementation process, as well as an unoptimized OpenMP version explicitly marking the major regions of parallelism. The focus on reimplementability also led us to generally choose benchmarks that were both interesting in structure, yet small enough for reimplementation of most or all of the benchmarks to be feasible. We expect that the entire benchmark set could be ported to another programming model in a few weeks by a small team of experienced programmers.

Since the simple versions are focused on simplicity and readability to support reimplementation, they should not be used as the basis of comparison for the other implementations in terms of performance. However, because the simple implementation does use the same general algorithm as the accelerated implementations, the autoparallelization community may see opportunities to transform the simple implementation directly to increase its performance. Such work is strongly encouraged, and may benefit from the other implementations as points of comparison.

The name “Parboil” comes from the culinary term for a partial cooking process that is typically followed with some finishing step before the dish is served. The term represents our belief that useful throughput computing benchmarks must be “cooked”, or released with a scalable algorithm suitable for a fine-grained parallel architecture. But we do not desire to provide only “fully cooked” versions, discouraging any modifications often necessary to demonstrate interesting research results. Nor do we desire to publish only “overdone” versions, completely optimized to a specific device’s low-level characteristics. The Parboil benchmarks should be fluid, with standard versions available for consistent points of comparison, but malleable enough to benefit or even reflect the in-flux state of the field at this time.

The remainder of this paper has three main objectives. The first objective is to acknowledge related work, contributions and influences on the Parboil benchmarks, and alternative benchmark suites. Ultimately, we want to help users make informed decisions about whether the Parboil benchmarks would be best used instead of or in combination with those other suites, which we accomplish in Section II. The second objective is to describe the Parboil benchmarks themselves, in Section III. The benchmark descriptions are intended to be a point of reference for those wishing to understand the benchmarks and their major computation patterns at some level without being forced to read the source code directly. We intend the descriptions to benefit primarily those intending to reimplement the benchmarks or those attempting to understand research results based on the benchmarks. The third and final objective of this document is to outline what we believe to be the best practices for publishing results based on the Parboil benchmarks, and encourage the research community to pursue the highest standards of research integrity. Finally, we recognize those whose collaboration and support made these benchmarks what they are today and offer some other concluding remarks in Section V.

II. Benchmark Suite Alternatives and Collaborations

Benchmarks are a prevalent part of the existing research work on computer systems, and many benchmark sets have been developed for different purposes. Depending on your own use case, Parboil may fulfill your requirements best alone or supplement other available benchmarks. You may also find that other benchmarks fully satisfy your needs. Here, we present what we know of other throughput computing benchmark suites available or in development to help you decide how Parboil might be most useful to you.

The initial Parboil benchmark set was created at the University of Illinois and used to demonstrate the performance and applicable code optimizations of the first-generation NVIDIA CUDA architecture [8]. However, the initial benchmarks were almost all some form of regular, dense array operation, which is only a small subset of the kind of applications that can benefit from throughput computing. The original Parboil benchmarks were also developed in 2008, when features such as atomic operations were absent from the GPU architectures available. In our revised Parboil benchmark suite, we include a few of the originals to maintain examples of dense array operations, but believe that the updated Parboil benchmarks represent a much more diverse benchmark set. The old Parboil benchmarks are therefore no longer fully supported, although the original distributions can be made available on request.

Multithreaded CPU performance has been the emphasis of several recent benchmark suites. The PARSEC benchmarks [1] are one such example, with a focus on broad coverage of characteristic workloads and parallelization patterns suitable for the shared-memory architecture of symmetric multiprocessors (SMPs) or chip multiprocessors (CMPs). They also embody an emphasis on consumer-directed recognition, mining and synthesis benchmarks. For those PARSEC benchmarks that would be applicable for studying accelerator architectures as well, we would

be interested in considering their inclusion into future versions of Parboil, likely with the current PARSEC implementation as a multithreaded CPU-optimized implementation. SPLASH and its successors are also applicable CMP benchmarks, but also have not been applied to throughput computing architecture studies. To the best of our knowledge, SPLASH has also not been actively updated in quite some time.[1] To a researcher most interested in comparing against previously published results for parallel shared-memory systems, reimplementing some of these benchmarks for an accelerator system may be most attractive. Indeed, we are aware that some efforts in this direction have begun [9].

The Rodinia benchmarks published by the University of Virginia [2] are very similar in philosophy and development to the Parboil benchmarks. They contain a similar mixture of building-block kernels and full applications, and also support both CUDA and OpenCL for most benchmarks. A researcher desiring a set of GPU computing source codes to experiment with, but that does not need any kind of uniform scripting environment for code multiversioning, collecting performance results, or changing platform settings across all benchmarks, may find the Rodinia benchmarks to be a better fit. The Rodinia benchmarks have no compilation complexity other than standard source code and Makefiles to understand. The support in Parboil provides easy ways of conducting certain kinds of research experiments at the cost of a slightly more complex compilation and execution system, although we try to make the standard interface user-friendly.

Arguably the largest force behind benchmarking in computing today is SPEC. The SPEC High-Performance Group is developing an accelerator benchmark suite, to which we are contributing the Parboil benchmarks as candidates. It is our suspicion that once that suite is published, many Parboil users may be able to migrate to the SPEC accelerator benchmarks, and gain the support of peer-reviewed results publishing. However, we also suspect that the Parboil development environment will likely prove more friendly to developing alternate implementations, especially if those alternate expressions require more than standard C, C++ or Fortran build systems.

III. Benchmark Summaries

The Parboil benchmarks themselves are a diverse set of applications. Some are intended to represent common library routines with broad applicability, such as SGEMM or sparse-matrix, dense-vector multiplication. Others are specific analyses applicable to a narrower domain, such as the two-point angular correlation function benchmark or MRI sample gridding. It would not be accurate to call any of the Parboil benchmarks a “complete application” in the typical sense, although several of the benchmarks are in fact run as standalone tools in some situations. Still, compared to end-to-end user applications, these benchmarks are relatively simple. It is an admitted weakness, but also a useful property in that a few graduate students could feasibly port the entire suite to a new programming model, or optimize them for a new architecture. A suite of full applications would practically prohibit such activities. We hope that in our explanation of the benchmarks their diversity and relevance will become apparent despite their “simplicity.”

Although the benchmarks include versions implemented in both the OpenCL and CUDA programming models, we will typically use OpenCL terminology in describing accelerated implementation details.

A. SGEMM

The SGEMM dense matrix operation is an important building block in numerical linear algebra codes. Packages such as the BLAS (Basic Linear Algebra Subprograms), one of the most commonly used numeric packages on dense matrices, are so important that major hardware vendors frequently provide their highly optimized BLAS packages such as MKL, ACML or CUBLAS. Because it is such a well understood application, often the first studied on any new architecture, and commonly the most heavily tuned operation on existing architectures, we include it in the Parboil benchmarks in spite of its “lack of novelty”.

In Parboil, the optimized CUDA versions for the NVIDIA architecture are based on the register-tiling GPU SGEMM code by Volkov [12], one of the fastest known implementations. However, unlike the implementation released by Volkov, the Parboil GPU kernel code has been re-implemented such that the readability, configurability, and performance of the code are better balanced. For example, certain code transformations like memory prefetching

[1]The distribution site for the most recent SPLASH benchmarks appears to be no longer valid. Although “cached” versions are available, we would appreciate any information about an active, primary source for the SPLASH-2 benchmarks.

and common subexpression elimination are adequately handled by current CUDA compilers; implementing them in source code reduces readability with little or no performance gain in our experiments. We also parameterize the code so that the degrees of register tiling/coarsening and shared memory tiling can be configured at compile time, greatly easing our efforts to tune the benchmark for different GPU generations. For example, although the code was originally designed for earlier generations of NVIDIA GPUs, with a simple adjustment of preprocessor macros setting tiling degrees, the implementation readily achieves over 500GFLOP/s on GTX480 GPUs.

B. Histogram (Histo)

The Histogram Parboil benchmark is a straightforward histogramming operation that accumulates the number of occurrences of each output value in the input data set. The output histogram is a two-dimensional matrix of char-type bins that saturate at 255. The Parboil input sets, exemplary of a particular application setting in silicon wafer verification, are what define the optimizations appropriate for the benchmark. The dimensions of the histogram (256 W $\times$ 8192 H) are very large, yet the input set follows a roughly Gaussian distribution, centered in the output histogram. Recognizing this high concentration of contributions to the histogram's central region (referred to as the "eye"), the benchmark optimizations mainly focus on improving the throughput of contributions to this area. Prior to performing the histogramming, the optimized implementations for scratchpad run a kernel that determines the size of the eye by sampling the input data. Architectures with an implicit cache can forego such analysis, since the hardware cache will automatically prioritize the heavily accessed region wherever it may be.

Two primary transformations reduce the amount of contentious updates. First, the eye region is privatized to batches of work-groups, the ideal number of batches determined experimentally for a given platform. The input is partitioned among the batches, each of which creates a partial private histogram, later reduced into the final output. These partial histograms can be generated in parallel, so all batches are launched together in one parallel kernel invocation.

Secondly, each batch is further decomposed into several thread blocks (work-groups), each assigned an exclusive region of the output in the batch's partial histogram. Each block in a batch handles the largest number of rows of the output histogram that still fits into its local memory, but all gather their input from the region assigned to their batch. Because the size of local memory is limited, every block can only compute a small portion of the histogram locally.

Therefore, if w is the maximum number of histogram rows that can be stored in local memory, and W is the size of the eye, it takes W/w blocks in each batch to generate the partial histogram from the batch's given portion of the input set. The input for the batch is read redundantly by each constituent block, which only processes the input falling within its tile. The relevant input is scattered into the local memory tile, a reasonably efficient operation. One designated block in the batch is also responsible for handling the rare inputs that fall outside the eye region, by atomically incrementing the appropriate bins in global memory. When blocks in a batch complete, their local tiles are juxtaposed with each other and the non-eye histogram regions in global memory, composing the entire partial histogram for the batch. The partial histograms are reduced into the final output.

Overall, the histogram benchmark demonstrates the high cost of random atomic updates to a large dataset. The global atomic update penalty can sometimes outweigh a fixed-factor cost of redundantly reading input data.

C. Cutoff-limited Coulombic Potential (CUTCP)

Some molecular modeling tasks require a high-resolution map of the electrostatic potential field produced by charged atoms distributed throughout a volume [4]. Cutoff-limited Coulombic Potential (CUTCP) computes a short-range component of this map, in which the potential at a given map point comes only from atoms within a cutoff radius of 12Å. In a complete application, this would be added to a long-range component computed with a less computationally demanding algorithm. In a simple, sequential implementation, each atom is visited in sequence, and electropotential contributions made by a visited atom are accumulated into all output cells within the cutoff distance before proceeding to the next atom.

In an accelerated implementation, the atom data is first presorted into a spatial data structure as follows. The atom-filled volume is partitioned into a 3D uniform grid of cells, and atoms are placed into a 3D array at an index determined by which cell they occupy in the space. The array has capacity for up to eight atoms in one cell; excess

atoms are processed separately on the CPU. For biomolecules, where the atom density is uniform and close to the density of water, this data structure utilizes memory efficiently. This presorting step is performed on the CPU.

In the accelerated kernel, an electrostatic potential value at a point is computed by scanning the contents of all cells within that point's cutoff radius, finding the atoms in those cells that are actually within the cutoff radius, and accumulating the potentials resulting from those atoms. In the most optimized versions, to reduce redundant computation, each thread computes the potential at multiple output points. To reduce memory bandwidth, one cell's worth of atom data at a time is loaded into local memory by a work-group, where it can be reused by multiple threads. All threads in a work-group scan the same set of atoms. This greatly reduces memory traffic at the cost of increased computation, since threads scan more atoms that do not contribute to the final calculation.

The optimized CUTCP application is compute-bound. Unlike the other compute-bound benchmarks, this kernel achieves high computational throughput partly at the cost of performing redundant computation. The percentage of redundant computation is the primary performance limiter for the hardware configurations we have studied.

D. Sum of Absolute Differences (SAD)

In video compression, motion is usually estimated by comparing a block of an input video frame to many candidate blocks in a reference video frame to find the most similar block. The Sum of Absolute Differences (SAD) benchmark represents the comparison step as used in the Joint Model (JM) reference H.264/AVC encoder's full-search motion estimation algorithm. The benchmark compares pairs of 4×4 pixel blocks taken from two different greyscale images, computing for each pair a number representing the difference between the blocks. In a separate step, it computes differences between larger blocks by summing the results for 4×4 blocks.

The computation for 4×4 pixel blocks dominates the parallel execution time. If we restrict our attention to a single block from the input frame, SAD is a 2D stencil algorithm. It is memory-intensive and benefits from caching as well as register tiling optimizations to exploit data reuse. The processing of multiple input frame blocks contributes an additional, outer level of parallelism. Blocks are permitted to straddle image boundaries, in which case out-of-bounds pixels have the same value as the nearest in-bounds pixel. To reduce the cost of bounds checking, boundary conditions are handled by texture sampling hardware on the GPU, and by precalculating a row of a block on the CPU. The GPU and CPU implementations differ in how loops were selected for register tiling. Coarsening was applied to reduce the number of costly texture accesses on the GPU, and register tiling to minimize the amount of arithmetic performed on the CPU.

Computation of SAD values for larger blocks consists of a sequence of vector additions. Additions are grouped into a few GPU kernels to reduce memory traffic.

E. Two-Point Angular Correlation Function (TPACF)

The Two-Point Angular Correlation Function (TPACF) is a measure of the distribution of massive bodies in space. The information of interest is a histogram of angular distances between all pairs of observed objects in space, as well as the relative value of that histogram distribution compared to the distribution of observed points correlated with a random set of points and the distribution of a random set correlated with itself. For robustness, one observed set is typically correlated with many random sets, each also correlated with itself.

To reduce the number of kernels launched, a generic kernel body is written that will correlate any two sets of data with a single thread block. The thread block performs tiling by iterating over both sets by tiles, one set cached into private registers and one cached into shared memory. Given two tiles, each thread computes the distance between its privately cached point with each point cached in shared memory. Contributions to the small histogram are highly contentious, so privatization is used to replicate the histogram, reducing contention. Multiple histogram copies per thread block are kept in shared memory, with threads hashed to histograms in a way minimizing contention. A reduction on the distributed histograms is the final step of the implementation.

F. Breadth-First Search (BFS)

The Breadth-first Search algorithm is commonly used in graph problems such as finding the shortest path between two nodes. The Parboil benchmark is credited to Luo et al. [5], and specifically optimized for a particular EDA application finding the shortest paths between a source node and every other node in a graph. In a sequential

implementation, the CPU takes every node in the current frontier and enqueues all unexplored neighbors to the next frontier. This process iterates until all the nodes in the graph have been visited.

In the parallel implementations, a task is created for every node in the current frontier. Each task explores all the neighbors of the node for unvisited nodes, and updates the next frontier accordingly. Each task adds a variable number of items to the work queue for the next frontier, which suggested a dynamic queue with atomic updates as the primary data structure. However, atomic updates to the variable representing the queue tail can be a major point of contention for a single, centralized queue.

To reduce the amount of contention when threads update the next frontier, multiple levels of privatization were applied to the queue and its tail, resulting in a hierarchical queue management system. At the lowest levels, threads are hashed into one of several local queues to reduce local memory contention. When the tasks finish discovering new frontier nodes, they merge the local queues into a compact, centralized queue for the group. Finally, the group commits the group queue contents to the global queue with one representative thread performing one atomic update to the global tail. The complexity of this parallel algorithm is O(V+E) with V being the number of vertices and E being the number of edges in the graph.

One interesting feature of the optimized implementation is its variety of different kernels for frontiers of different size. When the frontier is very small, only one work-group is created, which iterates through frontiers with fast, local synchronizations until the frontier is too large to cache in its local memory. When the frontier is too large to cache in one compute unit's local memory, but small enough to fit in the combined local memory of the entire device, a number of work-groups equal to the number of compute units is created to advance the frontier with global barrier synchronization among the concurrently executing work-groups implemented through atomic operations. This is the most efficient version of the kernel when the frontier just barely fits in the combined local memories, as it utilizes the whole chip's resources but rarely has to read and write queue entries to global memory. When the frontier is too large to be cached in the entire chip's resources, a third kernel based on the streaming model is launched to stream in the entire current frontier and write out the entire subsequent frontier to global memory. Thus, for very large graphs, most of the processing time is spent in the streaming kernel, and the fundamental performance bottleneck is memory bandwidth.

G. MRI Cartesian Gridding (Gridding)

Gridding is a preconditioning step that maps non-Cartesian MR samples onto a Cartesian grid before applying and taking the inverse Fast Fourier Transform of the sample space to generate an image. Gridding interpolates each input sample onto the grid points that fall within its vicinity using the Kaiser-Bessel function to determine the weight of the sample point onto each grid point based on the distance between them. Because the contribution becomes insignificantly small as the distance increases, it is approximated to zero beyond a certain cutoff, reducing the algorithmic complexity to only scale with the input size instead of the input/output size product.

For scalable parallel performance, gridding is best executed using a gather approach instead of the "natural" scatter approach, in order to eliminate write contention in the output. In addition, because the input has an unknown distribution[2], we sort it into bins before executing the gridding kernel. The biggest challenge of MRI gridding is the highly non-uniform distribution of its input data. The non-uniformity is a result of the acquisition method which collects far more points in the center of the space, where the most useful imaging information is found, than along its edges. Making all the bins equal in capacity would lead to a large inflation of the bin data structure due to padding elements. Instead, the binning kernel produces a compressed, sparse representation, in which each bin has variable capacity exactly matching the number of constituent elements. Accessing the compact bins is more challenging than accessing uniform-sized bins, requiring the precomputation of the starting offset of each bin. However, the bandwidth savings more than compensate for the additional complexity of access.

The gridding benchmark also enforces a cap on the capacity of each bin, and spilling over the excess elements to the CPU. This puts a hard limit on the amount of load imbalance that may afflict the gridding kernel, while also employing the host CPU's computational resources in parallel with the GPU gridding kernel.

[2]In practice, the distribution may in fact be "known", but may change after the gridding implementation is written or the MRI device purchased, and therefore must be considered unknown.

H. Stencil

The importance of solving partial differential equations (PDE) numerically as well as the computationally-intensive nature of this class of application have made PDE solvers an interesting candidate for accelerators. In the benchmark we include a stencil code, representing an iterative Jacobi solver of the heat equation on a 3-D structured grid, which can also be used as a buliding block for more advanced multi-grid PDE solvers.

The GPU-optimized version draws from several published works on the topic, containing a combination of 2D blocking [7] in the X-Y plane, and register-tiling (coarsening) along the Z-direction, similar to the one developed by Datta et al [3]. Even with these optimizations, the performance limitation is global memory bandwidth for current GPU architectures we have tested.

I. Sparse Matrix-Dense Vector Multiplication (SpMV)

Sparse matrix-vector multiplication is the core of many iterative solvers. SpMV is memory-bandwidth bound when the matrix is large. Thus, most optimization efforts have focused on improving memory bandwidth for both regular and irregular access.

Sparse matrix data can be stored or transformed into many previously studied data layout patterns, such as compressed sparse row (CSR) format, ELLPACK (ELL) format, and Jagged Diagonal Storage (JDS) format. Each is designed to store non-zero elements efficiently with its own regularization approaches. Computation in these three formats typically results in regular accesses over non-zero elements, both value and index fields, and irregular, data-dependent accesses over the dense vector. Considering which format to use for the GPU-optimized version, we note that JDS and ELL formats both work with finer parallelism granularities more easily than CSR format. Particularly, the JDS format is well-designed for parallel or vector processors, and can be viewed as a modification of ELL format minimizing imbalance among adjacent rows through row permutation. Because of its better load balance characteristics, we chose to base the GPU-optimized version on JDS format.

JDS format naturally results in stride-one access for the sparse matrix elements. Padding may be introduced to align data, but introduces holes in the input that become overhead bandwidth for a bandwidth-limited kernel, and is only applicable on certain architectures where alignment is very crucial. Other optimization efforts focused on the irregular accesses to the dense vector. On a GPU architecture without a general cache, the texture unit's cache can be used to improve the efficiency of irregular accesses to the vector data. Prefetching is also applied to hide more memory latency when high thread-level parallelism is not sufficiently available to hide latency alone.

J. Lattice-Boltzman Method simulation (LBM)

The Lattice-Boltzman Method (LBM) is a method of solving the systems of partial differential equations governing fluid dynamics [6]. Its implementations typically represent a cell of the lattice with 20 words of data: 18 represent fluid flows through the 6 faces and 12 edges of the lattice cell, one represents the density of fluid within the cell, and one represents cell type or other properties e.g., to differentiate obstacles from fluid. In a timestep, each cell uses the input flows to compute the resulting output flows from that cell and an updated local fluid density.

Although some similarities are apparent, the major difference between LBM and a stencil application is that no input data is shared between cells; the fluid flowing into a cell is not read by any other cell. Therefore, the application has been memory-bandwidth bound in current studies, and optimization efforts have focused on improving achieved memory bandwidth [13], [11].

To reach for better bandwidth, optimizations for GPU architectures focus on addressing the layout of the lattice data. The effective bandwidth delivered by GPU architectures is primarily determined by two factors: coalescing and bank conflicts. Coalescing is an attribute of a single instruction executed for many work-items. If the addresses accessed by those work-items are contiguous, then the memory system delivers a single DRAM burst to satisfy the "coalesced" accesses. Memory bank conflicts arise due to the distributed nature of the DRAM memory system. Different DRAM modules, or "banks" contain different sets of addresses. If the active threads in the GPU all happen to access different addresses in the same memory bank, the accesses will be serialized at that bank even as the other banks remain idle. The resulting memory bandwidth delivered is therefore much less than the peak bandwidth that could have been available.

The LBM benchmark base GPU version uses the most logical layout for software engineering: a large array of cell structures. Data layout transformation results in an optimized GPU version using a tiled structure-of-arrays

Benchmark	Typical Bottleneck of an Unoptimized Implementation	Optimizations Applied	Optimized Implementation Bottleneck	Potential Improvements
cutcp	Contention, Locality	Scatter-to-Gather, Binning, Regularization, Coarsening	Instruction Throughput	Minimizing Reads/Checks of Irrelevant Bin Data
mri-q	Poor Locality	Data Layout Transformation, Tiling, Coarsening	Instruction Throughput	
gridding	Contention, Load Imbalance	Scatter-to-Gather, Binning, Compaction, Regularization, Coarsening	Instruction Throughput	Minimizing Reads/Checks of Irrelevant Bin Data
sad	Locality	Tiling, Coarsening	Memory Bandwidth/Latency	Target Devices with Higher Register Capacities
stencil	Locality	Coarsening, Tiling	Bandwidth	
tpacf	Locality, Contention	Tiling, Privatization, Coarsening	Instruction Throughput	
lbm	Bandwidth	Data Layout Transformation	Bandwidth	
sgemm	Bandwidth	Coarsening, Tiling	Instruction Throughput	
spmv	Bandwidth	Data Layout Transformation	Bandwidth	
bfs	Contention, Load Imbalance	Privatization, Compaction, Regularization	Bandwidth	Avoiding Global Barriers / Better Kernels for Midsized Frontiers
histogram	Contention, Bandwidth	Privatization, Scatter-to-Gather	Bandwidth	Reducing Reads of Irrelevant Input (alleviated by cache)

TABLE I: Architecture Stresses of Benchmarks Before and After Optimization

implementation that achieves both good coalescing and parallel memory bank usage. The best tiling does depend on the specific GPU architecture. We provide a simple `layout_config.h` header file for easy exploration of different layouts for various architectures.

K. Mri non-cartesian Q matrix calculation (MRI-Q)

One of the original Parboil benchmarks, MRI-Q, calculates equation 3 in the GPU-based MRI reconstruction paper by Stone et al. [10], and is based on the implementation used to publish their work. An MRI image reconstruction is a conversion from sampled radio responses to magnetic field gradients. Sample "coordinates" are in the space of magnetic field gradients or *k-space*. The Q matrix in MRI image reconstruction is a precomputable value based on the sampling trajectory, the plan of how points in k-space will be sampled. The algorithm examines a large set of input representing the intended MRI scanning trajectory and the points that will be sampled. Each element of the Q matrix is computed by a summation of contributions from all trajectory sample points. Each contribution involves a three-element vector dot product of the input and output 3-D location, and a few trigonometric operations.

The output Q elements are complex numbers, but the inputs are multi-element vectors. An output element (and its corresponding input denoting its 3-D location) is assigned to a single thread. To make sure the thread-private data structures exhibit good coalescing, a structure-of-arrays layout was chosen for the complex values and physical positions of a thread's output. The shared input data set, however, is cached using GPU constant memory or some other high-bandwidth resource, and elects an array-of-structures implementation to keep each structure in a single cache line. When limited-capacity constant memory is employed, the data is tiled such that one tile is put in constant memory before each kernel invocation, which accumulates that tile's contributions into the output.

MRI-Q is a fundamentally compute-bound application, as trigonometric functions are expensive and the regularity of the problem allows for easy management of bandwidth. Therefore, once tiling and data layout remove any artificial bandwidth bottleneck, the most important optimizations were the low-level sequential code optimizations improving the instruction stream efficiency, such as loop unrolling.

L. Summary

Table I shows a brief summary of the major architectural features stressed by each benchmark in its unoptimized and optimized forms. The table highlights the ultimate effects of the optimization techniques applied to each

Benchmark	base	cuda-base	cuda-fermi	cuda-generic	ocl-base	omp-base
cutcp	X	X		X	X	X
mri-q	X	X		X	X	X
mri-grid	X	X		X	X	X
sad	X	X		X	X	X
stencil	X	X	X	X	X	X
tpacf	X	X		X	X	X
lbm	X	X		X	X	X
sgemm	X	X	X	X	X	X
spmv	X	X		X	X	X
bfs	X	X		X	X	X
histogram	X	X	X		X	X

TABLE II: Benchmark versions included as of February 23rd, 2012

benchmark, and our best assessment of the fundamental limitation for further performance improvement in the optimized implementation. Although each application is unique, we hope that this set at least presents a multifaceted view of the important characteristics of architecture and optimizations for general throughput computing.

Table II show the benchmark versions currently supported. We are committed to contributing more versions optimized for more architectures, and invite others to contribute optimized versions as well to build the base of optimized versions available for architecture comparisons. Furthermore, if any new or existing architectural characteristics not significantly probed by these benchmarks are brought to our attention, we would be glad to expand or adjust the Parboil set with new applications and algorithms to gain insight into those areas.

IV. Guidelines for Research and Publication

We envision three primary use models for Parboil. The first usage model is for calibrating the performance of real machines. To support this usage, we plan to provide a baseline performance for each benchmark. The guidelines and rules for such usage will be similar to that of the SPEC benchmark set. The details of the hardware, software stack, and code optimizations performed will be published at the Parboil web site. The system used to perform such measure will also be maintained and preserved at the Parboil hosting institution for at least five years. When a party reports a Parboil performance number for a system, the hardware, software stack, and code optimizations performed should be documented to the same level of details as the baseline number. After verification by the Parboil support team, the reported numbers, the system configuration, and the optimized code used for the publication will be published at the official Parboil web site. An organization similar to the SPEC consortium is being formed to support this usage model. We believe that with its guidelines, flexibility, and support, Parboil will stimulate healthy competition among vendors in this fast advancing area of throughput computing.

The second Parboil use model for Parboil is to evaluate new innovations in computer architecture. In this usage, we envision that researchers will use Parboil benchmarks in their simulation studies. Due to the research nature of such usage, we will not require a particular guideline but will work to foster an environment where researchers can more easily build on each others work. In particular, advanced research in future parallel architectures will most likely involve new program optimization techniques. We will encourage researchers to deposit their own optimized versions of Parboil benchmarks into a community repository. We will also encourage researchers to describe the simulation configuration, compilers and software stacks assumed in their experiments. The Parboil hosting institution will periodically compile an annotated list of research papers that use Parboil.

The third use model for Parboil is to study the benefit of new compilation and runtime techniques for real machines. We envision that researchers and developers will report enhanced Parboil numbers for real systems due to their new techniques. They will be encouraged to publish the details of the hardware, software stack, and code optimizations performed at the Parboil web site. However, the Parboil hosting institution will not verify the results. In particular, the Parboil hosting institution will not require the submitting teams to release their compiler or runtime implementations. Rather, the teams that submit these results are simply expected to adhere to conventional academic publication integrity guidelines. The intention is to foster an environment where researchers can make better comparisons by seeing the input code used in each others experiments. It will be up to the researchers and developers to share or release their actual implementation.

In all cases, we will remind users to respect the intended use of each version of the benchmark. For instance, the reference sequential version of a benchmark is primarily intended to facilitate reimplementation, not provide a point of performance comparison point based on CPU architectures. Researchers should not publish performance results of the reference sequential implementation, with an exception for authors specifically documenting performance improvements made to the reference sequential code itself through automatic parallelization or other techniques. Parboil results for CPU systems should be reported with CPU-optimized Parboil implementations using techniques such as cache tiling, multithreading, and vectorization with intrinsics or compiler directives as applicable, such as the examples we provide for MRI-Q and Histogramming.

The same consideration also applies to the baseline accelerated versions of each benchmark. This version is meant to provide researchers and developers with insight into the types of parallelization and optimization strategies that will likely be suitable for the corresponding benchmark. They may be used as points of comparison with architecture-optimized versions on the same system, but the versions one should use to compare two systems are the architecture-optimized versions.

Finally, we envision that the optimized versions of each Parboil benchmark program will evolve over time with community contribution. We encourage researchers and vendors to create new versions of benchmarks, especially those representing the state of the art in optimization for a particular programming model or platform and to submit their versions to be included in the publicly available repository. Copyright and licensing issues are addressed on the official Parboil web page:

`http://impact.crhc.illinois.edu/parboil.php`

We hope that, through mutually sharing our updates and contributions, the community can recognize the highest level of performance for each platform and credit contributors. Such collaborative and competitive efforts will also contribute to our understanding of parallel computing platforms as a community.

V. Conclusions

Although we have already acknowledged through citation most if not all of the external contributors for each of the benchmarks individually, we want to thank all our collaborators and sponsors explicitly and directly here as well. When appropriate, the acknowledgements are noted with the benchmark embodying their primary contributions. Our thanks go to John Stone, David Hardy, Klaus Schulten, and all the other members and collaborators of the Theoretical and Computational Biophysics Group at the University of Illinois (CUTCP); Ian Atkinson and Keith Thulborn of the Center of MR Research at the University of Illinois at Chicago (Gridding); Lijuan Luo and Martin Wong (BFS); and the multiple students of ECE 498 AL that allowed us to use their final projects as benchmark seeds. Finally, the members of the IMPACT research group not listed as direct authors have all made some meaningful contribution. Thanks especially to Xiao-Long Wu, Hee-Seok Kim, Timothy Wentz, Izzat El Hajj, Sara Baghsorkhi, Deepthi Nandakumar, Xiaohuang Huang, Stephanie Tsao, and Melvin Lathara.

This work is not directly supported by any particular grant or funding agency, but is supported in part by all the supporters of the IMPACT research group and its members. This work is supported in part by the following parties: the Gigascale Systems Research Center (GSRC), a center funded under the Focus Center Research Program of the Semiconductor Research Corporation (SRC); the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois, jointly funded by Intel and Microsoft; the National Science Foundation, under grant NSF CNS 05-51665; the National Institute of Health under grants NIF-P41-EB03631-16 and NIG-R01-CA098717; and NVIDIA Corporation.

We look forward to working with the throughput computing community to continue to maintain high standards for system benchmarking, and insightful analysis from architecture and optimization efforts targeted at these benchmarks. Please contact us if you would like to collaborate on the further development of the Parboil benchmarks.

References

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] D. J. Hardy, J. E. Stone, K. L. Vandivort, D. Gohara, C. Rodrigues, and K. Schulten. Fast molecular electrostatics algorithms on GPUs. In *GPU Computing Gems*. 2010.

[5] L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, pages 52–55, June 2010.

[6] Y. H. Qian, D. D'Humieres, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17(6):479–484, 1992.

[7] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. *SC Conference*, 0:32, 2000.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08*, pages 73–82, New York, NY, USA, 2008. ACM.

[9] M. Sinclair, H. Duwe, and K. Sankaralingam. Porting CMP Benchmarks to GPUs. Technical report, Department of Computer Sciences, The University of Wisconsin-Madison, 2011.

[10] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu., Z. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *International Conference on Computing Frontiers*, pages 261–272, 2008.

[11] I.-J. Sung, J. A. Stratton, and W. mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Conference on Parallel Architectures and Compilation Techniques*, pages 513–522, September 2010.

[12] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing '08*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[13] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *Visual Computing*, 24(5):323–333, 2008.