Workload Aware Algorithms for Heterogeneous Platforms

Kishore Kothapalli, Sivaramakrishna Indarapu, Shashank Sharma,
Dip Sankar Banerjee, Rohit Nigam
Center for Security, Theory, and Algorithmic Research
International Institute of Information Technology, Hyderabad
Hyderabad, India 500 032.
{kkishore@, {sivaramakrisha, shashank.sharma, dipsankar.banerjee, rohit_n}@research.}iiit.ac.in

***Abstract*— Algorithms that aim to simultaneously run on a heterogeneous collection of devices on a commodity platform have been in recent research focus. On such platforms, individual devices can have very differing architectures, clock rates, and execution models. Hence, one of the fundamental challenges in designing and implementing such algorithms is to identify load balancing mechanisms that aim to apportion the right amount of work for each device.**

The state-of-the-art in load balancing of heterogeneous algorithms has several drawbacks. Static solutions that partition the work irrespective of the input instance cannot lead to well-balanced load. On the other hand, analytical methods to identify the right work partition are available for only a few workloads or special cases of a few workloads.

In this paper, we propose a light-weight, low overhead, and completely dynamic framework that addresses the load balancing problem of heterogeneous algorithms. Our framework will be applicable for workloads which have a few simple characteristics such as having a collection of largely independent tasks that are easily describable. To show the efficacy of our framework, we consider two different heterogeneous computing platforms, and three different workloads: spmm, LBM, and ray casting. For each of the above workloads, we demonstrate that using our framework, we can identify the proportion of work to be allotted to each device up to $\pm 8\%$ on average. Further, solutions using our framework require no more than 5% additional time on average compared to best possible load assignment obtained via empirical search.

I. Introduction

Accelerator based computing using manycore architectures such as the GPUs and the IBM Cell BE has been successful in pushing application performance on commodity systems (cf. [35], [9], [34]). Further, GPUs have now become ubiquitous even on commodity systems due to their low price and low power consumption. However, it is projected that improvements in multicore CPU technology will further drive the performance of CPUs so as to scale challenges such as the power wall, the memory wall, and the like. Further, it is strongly believed that future generation computer systems shall be heterogeneous in nature with a mix of multicore CPUs and special purpose accelerators. Hence, it becomes important to design and implement efficient algorithms that work seamlessly on heterogeneous systems. This is termed as *heterogeneous computing* or *hybrid computing* in various recent works (cf. [3], [27], [39], [15], [2]). Heterogeneous computing has the scope to offer improved resource usage apart from faster algorithms.

However, heterogeneous algorithm design and implementations is fraught with a host of challenges. Current heterogeneous architectures impose severe limits on bandwidth available for communicating across the devices. This poses serious concerns to algorithm design and implementation. Synchronization between devices poses yet another difficulty. Therefore, at present, designing and implementing heterogeneous algorithms comes with a lot of hand-crafting.

Nevertheless, heterogeneous computing on commodity platforms is gaining large scale research attention in recent years. Such algorithms have been designed recently for several challenging problems in parallel computing including graph BFS [11], [27], [19], dense matrix computations [43], sorting [4], and the like. Many of the above-cited works spread the entire computation across the computational devices. In some cases, this is followed by a post-processing phase that combines the outputs of the individual computations [3], [27], [4], [36], [39]. This approach of designing heterogeneous algorithms can be called as the *work partitioning* approach.

However, due to differing architectures and execution characteristics, the amount of work each device can do in a given computation can vary significantly across the devices. Therefore, using the work partitioning approach in the design and implementation of heterogeneous algorithms brings up the problem of load balance across devices. As heterogeneous algorithms should aim to improve the resource usage, it is required that proper load balance mechanisms are deployed.

Current approaches to address this problem include either a static work partitioning or using an analytical model to determine the appropriate work partitions. In the former case, one typically uses a standard dataset for a given workload and arrives the proportion of work allocation. This approach, used in [27], [3], [39], [15] to name a few, is very limiting as it fails to capture any properties of a different instance before deciding the work partition. In the latter case, one proposes an analytical model that captures the time taken by each device on a given input. This model can then be used to identify the work proportion. This approach is used in [27, Section IV.C]. However, the latter approach is not generic and is known to work only for very specific problems or specific input instances of a problem [27]. Thus, existing approaches fall short of a complete solution.

Thus, a fundamental problem in heterogeneous computing is to propose generic mechanisms that can help address the issue of load balancing in heterogeneous algorithms designed using the work partitioning model [13]. In this paper, we propose a simple and light-weight mechanism for the same. Our mechanism has the special property that it can offer load balancing even under dynamic conditions, and is suitable for a variety of application workloads which exhibit some common characteristics. We also validate our proposal against three different workloads: sparse matrix multiplication (`spmm`), Lattice Boltzman Method (**LBM**), and Ray Casting (**RC**). These workloads are chosen for their wide-ranging applications and importance. These workloads have been part of several studies on benchmarks and are also included in the recent highly-influential work of Lee et al. [25].

We note that our approach can be applied to design heterogeneous algorithms for other workloads also. Our work thus indicates that efficient dynamic load balancing approaches can be designed with relatively little additional effort. While we focus on CPU+GPU heterogeneous computing platforms in this paper, our framework described in Section III can be easily adapted to other heterogeneous computing platforms.

A. Related Work

One of the works on dynamic task management was proposed by Song et. al. in [36]. In this paper, the authors describe a dynamic task scheduling system for distributed memory systems which does not have any dependence on process cooperation. They design a dynamic runtime system for linear algebra libraries like PBLAS and ScaLAPACK and uses algorithms to resolve data dependencies without process cooperation. They test their solution on three well known linear algebra routines such as the Cholesky Factorization, LU Factorization and QR Factorization.

In the field of heterogeneous systems, most of the dynamic task scheduling work has focused on shared memory systems. One of the early works that was proposed was in Cilk [33]. Cilk is a generalized multithreaded language that uses the semantics of C. It uses a ”work-stealing“ algorithm to schedule the tasks. The Cilk model has found a large number of applications in solving recursive problems like the ones proposed in [6], [14].

Another work on fine grained dynamic task allocation was proposed by Buttari et al. in [23]. The work of [23] focuses on linear algebra applications and uses a block data based layout for scheduling the tasks dynamically. That same work has been now extended to form the PLASMA library [32] that provides a more enriched set for linear algebra algorithms.

SMP Superscalar [8] is a programming environment proposed by the Barcelona Supercomputing Center for multicore architectures. The SMP system compiles pieces of C code and links it with a runtime system. The runtime system then runs the program in parallel. It is similar to the TBLAS where the SMP system replaces the sequential linear algebra routines using a task-based library and runs them in parallel.

Workload	Baseline	Dataset	Hetero-High		Hetero-Low	
			Split %	Runtime	Split %	Runtime
`spmm`	[27]	[42]	5%	6%	7%	11%
`csrmm`	[27]	[42]	4%	5%	5%	7%
RC	[26]	[26]	1%	2%	3%	8%
LBM	[16]	**uar**	1%	2%	1%	2%

TABLE I

TABLE SHOWS THE RESULTS OF OUR IMPLEMENTATIONS WITH RESPECT TO BASELINE IMPLEMENTATIONS ON TWO HETEROGENEOUS PLATFORMS, LABELED ”HETERO-HIGH” AND ”HETERO-LOW” AND DESCRIBED IN SECTION II. THE PHRASE **uar** REFERS TO DATASETS GENERATED UNIFORMLY AT RANDOM.

There are other works such as Lan et al. [24] and Horton [20] which propose a dynamic load balancing schemes for parallel systems. However, these are limited to multicore systems, and does not address heterogeneous systems. Similar strategies for grid based systems are proposed in [17].

In our work we mostly focus on work balancing on commodity heterogeneous systems that are tightly coupled. Our work focuses on more future generation systems, as it is anticipated that computer architectures in the near term are expected to the heterogeneous, and possibly on a single die. educed.

B. Our Results

Our main result is the design of very efficient and light-weight strategies for dynamic load balancing of tightly coupled commodity heterogeneous platforms. To this end, we propose a largely lock-free, completely dynamic, low overhead, and light-weight framework. Our framework is also extendible to other heterogeneous computing platforms with minimal changes. Further, our framework identifies characteristics of workloads to which the framework can be applied.

We also show three case-studies: `spmm`, **RC**, and **LBM** to validate the framework. A summary of our results over two different CPU+GPU heterogeneous computing platforms is presented in Table I. All the comparisons are performed with respect to the corresponding best reported heterogeneous baseline implementations that use a static work partitioning strategy to identify the best possible work splitting ratio. The phrase *Split %* (*Runtime*) refers to the absolute difference in the work partition percentage (*resp.* runtime) of our implementation compared to that of the baseline implementation. These results in each column are averaged over the dataset under consideration over multiple runs.

C. Organization of the Paper

The rest of the paper is organized as follows. Section II describes the computing platforms used in our experiments, and also describes the workloads we use in validating our framework. Section III describes our framework in a generic manner. Section IV-VI show how our framework can be used for the three different workloads along with results on the various computing platforms. The paper ends with some concluding remarks in Section VII.

II. An Overview of our Heterogeneous Computing Platforms

In this section, we briefly describe the heterogeneous computing platforms used in our experiments. These platforms are a combination of Intel multicore CPUs and Nvidia GPUs. We use two different heterogeneous platforms for conducting the experiments. One is a high end platform, labeled *Hetero-High* which is typically the one that is used for developmental purposes and is composed of server class hardware. The other platform, labeled *Hetero-Low*, is a representative of commonly used commodity desktop and laptops configurations. We specifically choose to use these two platforms in order to show the applicability of our framework on a wide spectrum of heterogeneous platforms. In Table II, we list the important characteristics of the two platforms we use in our experiments.

1) Hetero-High: The Hetero-High heterogeneous platform we used is a coupling of the two devices, the Intel i7 980 and the Nvidia GTX 580 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1. The CUDA API Version 4.1 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution.

The GTX 580 GPU is a current generation Fermi micro-architecture from NVidia that has 16 symmetric multi-processors (SM) with each SM having 32 cores for a total of 512 compute cores. Each compute core is clocked at 1.54 GHz. Each SM has a hardware scheduler that schedules 32 threads at a time. This group is called a *warp* and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU now has a fully cached memory access via an L2 cache, 768 KB in size. In all, the GTX 580 has a peak single precision performance of 1.5 TFLOPS.

Along with the GTX 580, we use an Intel i7 980x processor as the host device. The 980x is based on the Intel Westmere micro-architecture. This processor has six cores with each core running at 3.4 GHz and with a thermal design power of 130 W. With active SMT(hyper-threading), the six cores can handle twelve logical threads. The L3 cache has a size of 12 MB. The L1 cache size is 64 KB per core and the size of the L2 cache is 256 KB. Other features of the Core i7 980x include a 32 KB instruction and a 32 KB data L1 cache per core and the L3 cache is shared by all 6 cores. The i7 980 CPU has a peak throughput of 110 GFlops.

2) Hetero-Low: Our low-end heterogeneous platform resembles a commodity desktop computing environment more closely. The Hetero-Low platform is a combination of an Intel Core 2 Duo E7400 CPU along with an NVidia GT520 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1, and we use OpenMP specification 3.0 to program the CPU along with ANSI C.

The GT520 is a stand-alone graphics processor having 48 computing cores and 1 GB of global memory. Each of the compute cores are clocked at 810 MHz. The GPU on an average give a sustained performance of 77.7 GFLOPS and consume about 29W of power. In this system both the processors are of a comparable performance range and hence provide a more realistic platform for experimenting the heterogeneous programs.

The Intel Core 2 Duo CPU is one of the earliest multicore offerings from Intel and was released in the year 2008. It has 2 cores with hyper-threading and each core is clocked at 2.8 GHz. The CPU consists of a 3 MB L2 cache and the maximum power consumption is around 65 W. The CPU was designed entirely for commodity PCs and gives a sustained performance of about 20 GFLOPS.

III. Our Framework

In this section, we present a generic description of our framework. To this, end, we first consider a few characteristics of workloads for which our framework is suitable. The characteristics we seek are:

- Independent work units: It must be the case that the computation can be broken down into independent subproblems. We call these subproblems as *work units*. It is not necessary that the work units have identical computational requirement.
- Easily describable work units: We seek workloads where it is easy and succinct to describe independent subproblems. For instance, a work unit could correspond to processing a contiguous set of elements, say rows in a matrix.
- Minimal or no post-processing: We seek that the solution to the entire problem be a (near)-immediate consequence of the solutions to the independent work units. So, there should be little post-processing involved.

Some of these characteristics are similar to divide-and-conquer or partitioning design methodologies. Parallel algorithm design has also exploited the above properties to design efficient algorithms (cf. [22]). Using our framework, we also show that algorithm engineering of heterogeneous algorithms can benefit from the above characterization.

We now describe our framework in the context of heterogeneous CPU+GPU systems for ease of exposition. Our proposed solution for dynamic load balancing of the above category of workloads envisages that the multiple threads of the CPU and the GPU share a queue that contains several work units. The individual threads can access the work queue to fetch the next work unit for which computation is still pending. We also have CPU threads and the GPU access the work queue from either end so that there is no need to, in most cases, synchronize accesses to the work queue by the CPU and the GPU. However, CPU threads have to access the queue in a

GPUs						
Device	**Cores**	**# of SMs**	**Clock**	**Global Memory**	**L2 Cache**	**Threads per block**
GTX580	512	16	1.54 GHz	1535 MB	768 KB	1024
GT520	48	1	1.62 GHz	1024 MB	64 KB	1024

CPUs						
Device	**# of Cores**	**SIMD width**	**Clock**	**Last Level Cache**	**L1 Cache**	**# of Threads**
i7 980x	6	4	3.4 GHz	12 MB	32 KB	12
Core 2 Duo	2	2	2.8 GHz	3 MB	32KB	4

TABLE II

THE SPECIFICATIONS FOR THE DIFFERENT GPUS AND CPUS USED IN OUR EXPERIMENTS.

concurrent fashion. Given the low number of CPU threads that one can use, we employ a simple locking mechanism on the CPU `front` variable. See also Figure 1 for an illustration.

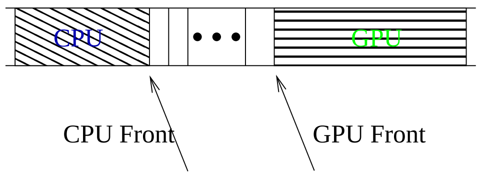


Fig. 1. Work queue showing the CPU and the GPU front pointers. The shaded region at both ends corresponds to the work processed by the CPU and the GPU respectively.

Note from our description that it is likely that the CPU and the GPU `front` pointers move at different rates. This indicates that the size of the work unit for a CPU thread and a GPU kernel can vary depending on the workload. The ideal size of the work unit for a CPU thread and a GPU kernel can be estimated based on the nature of the workload and other parameters. Using our framework requires one to address a few optimizations for improved performance. Some of these are mentioned below.

- Minimal Synchronization: Firstly, we seek to minimize the synchronization requirement when accessing the queue. For this, we make the queue double-ended. The CPU and the GPU dequeue work units from either ends. The only synchronization required between the CPU and the GPU is for dequeuing the last work unit from the queue. Having a double ended queue also ensures that it is easy to maintain the state of the queue correctly at all times. In practice, we also notice that the synchronization requirement is also most non-existent. This optimization is possible for workloads that possess the characteristic that work units are independent.
- Reducing the Overhead of Queue operations: Secondly, the overhead of queue operations should be kept at a minimum so that heterogeneous algorithms using our framework have a runtime as close to the best possible runtime. For this, we envisage that the queue can actually be maintained only logically, and not physically. So, one need not actually fill the queue with work units before the start of the computation. Initially, the `front` pointer on the CPU side is at work unit 1, and the `front` pointer on the GPU side is at work unit n, assuming there are n work units in total. The progress of the computation can be described by storing the location of the queue `front` pointer at each end. The computation is said to finish when the `front` pointer on the GPU side meets, or crosses, the `front` pointer on the CPU side. This optimization is possible whenever the workload has the characteristic that work units are succinctly describable.
- Other Program Optimizations: We also introduced several optimizations in implementing our framework. For instance, it is well known that there is a small overhead associated with launching a GPU kernel from a host device. To keep this overhead low, we actually launch the GPU kernel only once irrespective of the number of work units that the GPU works. The GPU kernel interacts with the host to fetch multiple work units without exiting execution on the device.

A. *Extensions*

While we have described the framework for CPU+GPU based heterogeneous systems, our framework can be extended to other settings also with minimal changes. For instance, one can apply our framework to other heterogeneous platforms with minimal modifications. What we need is program support for handling the interaction between programs executing on different devices in the platform.

We note that our framework can be applied to cluster based systems also. In this case, we have to adapt a one-dimensional work queue to a multi-dimensional grid. Work units in the computation will now correspond to points on multi-dimensional grid. If the workunits are independent of each other, then such a correspondence between points on a multi-dimensional grid and the work units can be easily created. This is akin to the standard GPU/OpenCL programming model of a grid.

IV. CASE STUDY 1 – SPARSE MATRIX MULTIPLICATIONS

Multiplying a sparse matrix with another sparse/dense matrix is an important workload in parallel computing. These operations, called `spmm` and `csrmm` respectively, have applications to several problems from varied domains. For instance, `spmm` has applications to problems from engineering such as graph algorithms [45], numerical applications including climate modeling, molecular dynamics, CFD solvers, and so on. `csrmm` is widely used in Krylov subspace methods such as Lanczos method and conjugate gradient method [12]. Indeed, sparse matrix operations are listed as one of the seven dwarfs in the Berekely report [1].

A. Sparse Matrix–Sparse Matrix Multiplication (`spmm`)

Let A, B be sparse matrices and $C = A \times B$, where A,B and C are the matrices of sizes $M \times N$, $N \times P$ and $M \times P$ respectively. Some of the recent approaches to provide efficient and scalable algorithms for `spmm` include [7], [27]. Often, it is a question of how to arrange the matrices in suitable data structures so that the expensive nature of highly irregular memory access patterns can be partly mitigated. It is shown in [27] that the *Row-Row* method, described below, is most suited for GPUs and also for CPU+GPU heterogeneous platforms. In the Row-Row method, the ith row in C, $C(i,:)$, is obtained by multiplying each element in $A(i,:)$ with the corresponding row in B. We then add all the scaled B rows to get the $C(i,:)$. In other words, $C(i,:) = \sum_{j \in A(i,:)} A(i,j) \cdot B(j,:)$.

As identified in [7], [27], one of the main challenges of arriving at an efficient algorithm for `spmm` is the difficulty in estimating the size of the output for a given input or a subset of a input. This suggests that the volume of computation for a subset of the input can vary substantially. However, when one uses the Row-Row formulation, the computation for each row of the output is entirely independent of the computation for other rows of the output. Further, a work unit in this case can be succinctly described as a contiguous set of output rows. Additionally, there is no post-processing involved. All these characteristics of the `spmm` workload indicate that the framework from Section III is suitable for `spmm`. In this section, we show that it is indeed the case.

B. Algorithm

Our heterogeneous algorithm for `spmm` can be described briefly as follows. Let $cpuSize$ and $gpuSize$ denote the work unit sizes of CPU and GPU respectively. Let $cpuOffset$ and $gpuOffset$ are two global variables to track working units of CPU and GPU. $cpuRows$,$gpuRows$ denotes the number of rows computed in that call made by CPU and GPU respectively. The GPU and CPU use Algorithm 1, Algorithm 2 respectively. In our implementation, the function $spgemmGPU$ uses the Row-Row based matrix multiplication from [27]. On the CPU side, the function $spgemmCPU$ uses the Intel MKL routine [21]. This algorithmic approach is used also for the other workloads in Section V–VI.

C. Sparse Matrix–Dense Matrix Multiplication (`csrmm`)

Let A be a sparse matrix, B be a dense matrix stored in the row major format, and let $C = A \times B$. Similar to `spmm`, `csrmm` also has all the characteristics of framework mentioned in Section III. So we can use work queue model for `csrmm`. The algorithm we follow is similar to the one mentioned in IV-B.

It is shown that our GPU implementation of `csrmm` from [27] outperforms the `cusparse` library implementation [31]. Hence, in our implementation, we use the implementation from [27] on the GPU and use the corresponding routine from the Intel MKL library [21] `csrmm` on the CPU.

Algorithm 1 Work queue model on GPU side

```
$cpuOffset = 0$; // intialized globally
$gpuOffset = M$;//intialized globally
while $cpuOffset < gpuOffset$ do
  if $cpuOffset < (gpuOffset - gpuSize)$ then
    $gpuRows = gpuSize$;
    $gpuOffset = gpuOffset - gpuRows$;
  else
    $gpuRows = gpuOffset - cpuOffset$;
    $gpuOffset = cpuOffset$
  end if
  $spgemmGPU(A, B, C, gpuOffset, gpuRows)$;
  Tranfer partial output asynchronously.
end while
```

Algorithm 2 Work queue model on CPU side

```
$cpuOffset$ =0; // initialized globally
$gpuOffset$ = $M$;//initialized globally
while $cpuOffset < gpuOffset$ do
  if $(cpuOffset + cpuSize) < gpuOffset$ then
    $cpuRows = cpuSize$;
    $cpuOffset1 = cpuOffset$;
    $cpuOffset = cpuOffset + cpuSize$;
  else
    $cpuRows = gpuOffset - cpuOffset$;
    $cpuOffset1 = cpuOffset$;
    $cpuOffset = gpuOffset$
  end if
  $spgemmCPU(A, B, C, cpuOffset1, cpuRows)$;
end while
```

D. Results

To validate our approach, in our experiments we have used the popular dataset of sparse matrices from the work of Williams et al. [42] which is shown in Table III. The dataset from [42] consists of 14 matrices from a wide range of applications areas and has been the choice dataset for works on sparse matrix operations in recent times [27], [28], [29], [30], [44].

In our experiments, the baseline algorithm is the heterogeneous algorithm from [27, Section III-A]. This baseline algorithm uses an empirical strategy to identify the best possible work distribution for the CPU and the GPU. As CPU and GPU have different architectures and computational models, $cpuSize$, $gpuSize$ are also different. In our experiment we varied both $cpuSize$, $gpuSize$ on a wide range of values to find the best values.

In Figure 2(a), Figure 2(b), we show the absolute difference in the work split percentage between our algorithm and the baseline algorithm for `spmm` and for `csrmm` respectively. This difference is calculated as the absolute difference between the work split percentage of the baseline implementation and our implementation. As can be seen, the average absolute difference in the work split percentage of our implementaiton

Matrix	Rows	NNZ	NNZ/Row
Dense	2,000	4,000,000	2000.0
Protein	36,417	4,344,765	119.3
FEM/Spheres	83,334	6,010,480	72.1
FEM/Cantilever	62,451	4,007,383	64.1
Wind Tunnel	217,918	11,634,424	53.3
FEM/Harbor	46,835	2,374,001	50.6
QCD	49,152	1,916,928	39.0
FEM/Ship	140,874	7,813,404	55.4
Economics	206,500	1,273,389	6.1
Epidemiology	525,825	2,100,225	3.9
FEM/Accelerator	121,192	2,624,331	21.6
Circuit	170,998	958,936	5.6
Webbase	1,000,005	3,105,536	3.1

TABLE III

LIST OF SPARSE MATRICES. NUMBER OF COLUMNS AND ROWS ARE EQUAL FOR ALL THE INSTANCES.

with respect to the baseline implementation is under 6% on both the heterogeneous platforms that we used in our experiments and for both `spmm` and `csrmm`.

To measure the time overhead of our framework, we measured the absolute difference in time taken by our implementation with respect to the baseline. The results of this comparison on two different heterogeneous platforms from Section II is shown in Figure 3(a) for `spmm`, and in Figure 3(b) for `csrmm`. As can be seen, the average absolute difference in the runtime of our algorithm with respect to the baseline algorithms is under 10% for both the workloads on both the platforms.

The slight difference in the work split percentage and the runtime of our implementation compared to the baseline implementation can be attributed to the overheads involved in our framework such as breaking the computation into several work units. On the other hand, the baseline implementation treats the portion of its work as a single work unit.

Finally, we also measured the percentage of time either device, the CPU of the GPU, is idle in our implementation. The maximum of these two can be called as the *idle time* of an implementation. This notion of idle time is an important metric in load balancing [24]. Both our `spmm` and `csrmm` implementation achieve a very low idle time of under 3% on average on the dataset used in our experiments.

V. CASE STUDY 2 – RAY CASTING

Ray Casting is an important visual application, used to visualize 3D datasets, such as CT scan data used in medical imaging. High quality image generation algorithms, known as ray casting, cast rays through the volume, performing compositing of each voxel into a corresponding pixel, based on voxel opacity and color. Since all rays perform the computations independently, the problem is very much portable for parallel architectures. Tracing multiple rays using SIMD is challenging because rays can access non-contiguous memory locations resulting in incoherent and irregular memory accesses.

One of the recent works for ray casting on modern architectures is that of Lurig et al. [26]. The algorithm is note-worthy for its simplicity and portability to parallel architecture. The algorithm from [26] can be divided into three stages. In the first stage of the algorithm, an interpolation function is calculated for each tetrahedron. In the second stage, ray entrance calculations are performed for the rays to be traversed. In the final stage, the rays are then traversed through the polygon to accumulate color and intensity values that are used for final rendering.

In our work, we redesign the algorithm of [26] for a heterogeneous CPU+GPU platform. A brief description of our algorithm is as follows. A preprocessing step is performed before the algorithm. The preprocessing stage generates the complete triangle list from the tetrahedron data, marks the surface triangles and removes the backfacing triangles from surface triangles to generate list during the second stage.

The calculation of the interpolation function is performed for each tetrahedron in the dataset, in parallel on GPU and CPU. The interpolation function is of the following form:

$$f(x, y, z) = a + bx + cy + dz$$

where (x, y, z) are the coordinates of any point inside the tetrahedron and a, b, c and d are the coefficients of the interpolation function which are being calculated.

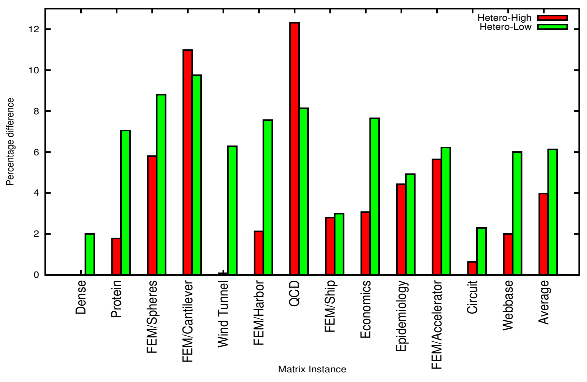
Each vertex of a polyhedron has an intensity value associated with it. These values are then used to generate the interpolation function by solving 4 linear equations using the Cramer's Rule.

Ray first-hit stage generates the list of the closest triangle intersected by each ray. This is required in the next stage when traversing the complete tetrahedron mesh. The step is performed in parallel over all the rays. The triangle data is loaded in the shared memory in the GPU implementation used by all the rays to perform intersection calculations.

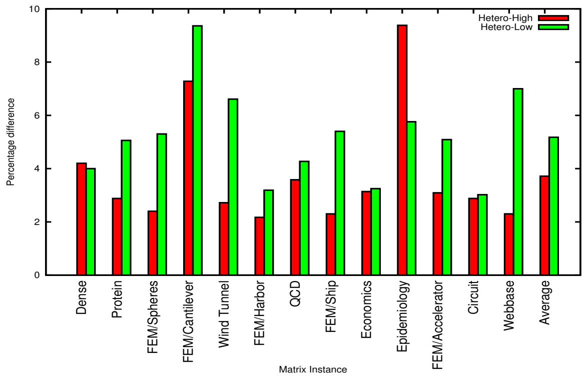
Upon finding the first intersection with a tetrahedron, the ray is traversed in front to back manner through the tetrahedra mesh. We represent a ray in its parametric form, $r = o + t\hat{d}$ where o is the origin of the ray and $\hat{d}$ is normalized direction vector. The ray is checked for intersection with rest of the 3 triangles of the current tetrahedron except the ray entrance triangle. The test outputs the next triangle getting intersected and also the t parameter value, which is then used to find the exit point of the ray for the tetrahedron. The intensity value for a tetrahedron is found by integrating the values given by the interpolation function at the start and end point of a tetrahedron for each ray. The integration is performed using the trapezoid rule. The process is repeated until the ray exits the mesh.

We note that the computation for each ray is indeed independent of the computation with respect to other rays. Further, a work unit in this case may correspond to the computation involved across a bunch of rays. This work unit can also be described succinctly by indexing the starting and ending ray numbers. Also, there is no post-processing involved at the end. Thus, ray casting satisfies the characteristics of workloads described in Section III.

We use the algorithmic model described in Section IV. A size of the work unit for the CPU and the GPU is chosen empiricially. We kept the size of the GPU work unit to be

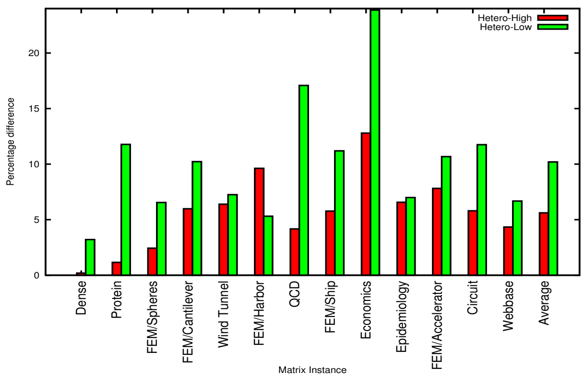


(a) `spmm` Split %

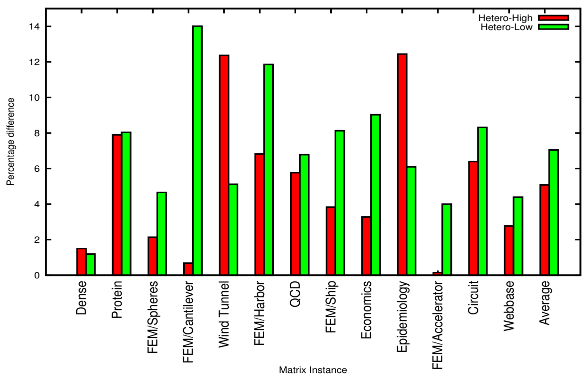


(b) `csrmm` Split %

Fig. 2. Figure shows the absolute difference in the work split percentage with respect to the baseline implementation for `spmm`; and `csrmm`. The last instance "Average" shows the average value of the series.



(a) `spmm` Time



(b) `csrmm` Time

Fig. 3. Figure shows the absolute difference in the runtime with respect to the baseline implementation for `spmm`. The last instance "Average" shows the average value of the series.

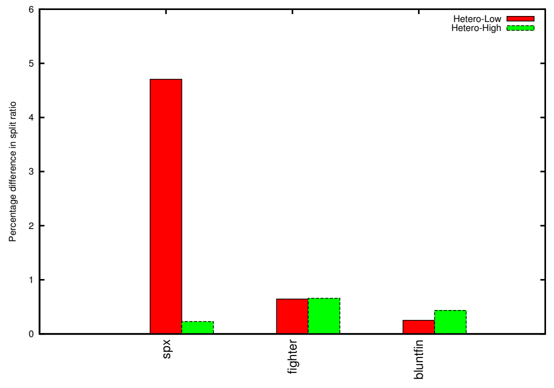
six times more than that of the CPU work unit size. This arrangement leads to lesser number of kernel calls while keeping both the CPU and the GPU busy till the very end. The number of rays used depends on the input image. Let the rays be numbered from 1 through n. Applying the framework from Section III, the computation corresponding to rays with indices starting from 1 is assigned to the CPU. Computation corresponding to the rays with indices from n backwards is assigned to the GPU. Since we use a multicore CPU, a block of rays allotted to the CPU is divided among the CPU threads equally. The CPU threads wait for the master CPU thread to do this work distribution. This reduces the synchronization overhead at the work queue. However, due to the waiting time of CPU threads, the overall time taken by our algorithm is slightly larger than the time taken by a heterogeneous algorithm that uses a brute-force method to identify the work split percentage and then uses such a work distribution.

A. Results

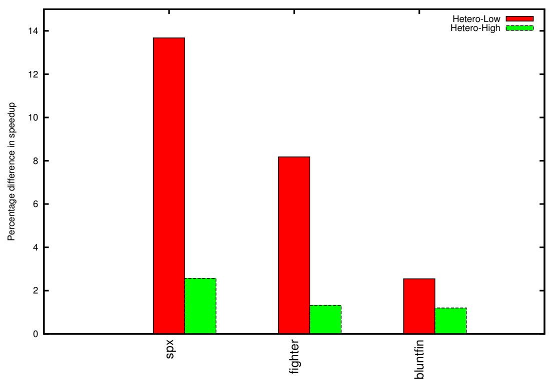
We use three different images: spx, fighter and bluntfin in our experiments. These input images have been part of the standard inputs used in most of the recent works on volumetric ray casting. The total number of tetrahedrons vary greatly among the three images and hence the results provide a fairly good overview of the algorithm. For the spx, the total number of tetrahedrons is $12,936$, the fighter image has $70,125$ tetrahedrons, and the bluntfin image has $222,414$ tetrahedrons.

The baseline implementation we use is a heterogeneous implementation of the algorithm of Lurig et al. [26]. The baseline implementation identifies the best work split percentage empirically. The results are reported for 512×512 resolution, with 10-15% screen coverage. The results are averaged over multiple iterations on two different heterogeneous platforms as described in Section II.

Figure 4(a) shows the difference in the work split percentage achieved by our implementation with respect to the baseline implementation. As can be seen, the difference in under 5% on all the three images. Figure 4(b) shows the difference in the runtime between our implementation and the baseline implementation. The average idle time of our implementation is also under 2% on both platforms. These results also indicate the applicability and efficacy of our model.



(a) Split %



(b) Time

Fig. 4. Figure shows the absolute difference in the split percentage runtime with respect to the baseline implementation for **RC**.

VI. Case Study 3 – Lattice Boltzman Method

Lattice Boltzman Method, **LBM** for short, is a class of computational fluid dynamics for fluid simulation. It is a numerical method to solve the Nevier-Stokes Equation for incompressible Newtonian fluids [10], [37]. The method is applied on a limited number of particles which simulates their streaming and collision. The intrinsic particle interaction leads to a good study of flow behavior applicable across the greater mass. We use the LBM D3 Q19 model to simulate fluid flow around a solid sphere. This has a three dimensional fluid lattice and each lattice point has 19 speed vectors as shown in Figure 5. LBM is done in two steps namely collision and streaming.

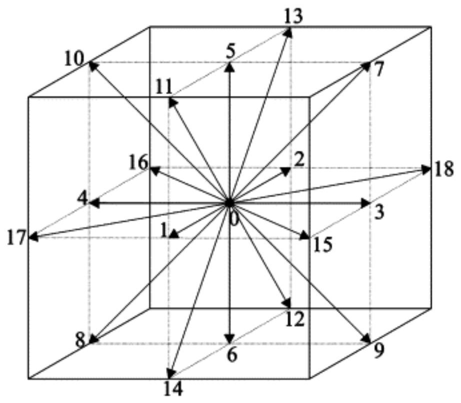


Fig. 5. A three dimensional lattice with 19 speed vectors. This model is called D3 Q19. This figure is taken from [40].

For each particle, let f_i^t represent the speed of the particle at time t in the ith direction. During the collision step, the speed of the particle at time $t + \delta t$ is computed as:

$$f_i^t(\vec{x}, t + \delta t) = f_i(\vec{x}, t) + \frac{1}{\tau}(f_i^{eq} - f_i), \text{ for } i = 0, 1, \cdots, 18.$$

The second term in the right hand side of the above equation is referred to as the BGK collision term [5]. Similarly, streaming at time $t + \delta t$ and position $x + \delta x$ is given by

$$f_i(\vec{x} + \delta x, t + \delta t) = f_i^t(\vec{x}, t + \delta t), \text{ for } i = 0, 1, \cdots, 18.$$

In the above equations, τ is the relaxation rate which relates to fluid viscosity (η), and is computed as $\eta = \frac{1}{3}(\tau - \frac{1}{2})$. f_{eq} represents the equilibrium values which approximate Maxwellian distributions. This is calculated as follows:

$$f_i^{eq} = w_i \cdot \rho \cdot \left(1 + 3(\xi_{\mathbf{i}}.\mathbf{u}) + \frac{9}{2}(\xi_{\mathbf{i}}.\mathbf{u}) - \frac{3}{2}(\mathbf{u}.\mathbf{u})\right)$$

where ρ is the density calculated as $\rho = \sum_{i=0}^{Q-1} f_i$. Further, we have $w_0 = \frac{1}{3}$, $w_1, \cdots w_6 = \frac{1}{18}$, and $w_7, \cdots, w_{18} = \frac{1}{36}$, [18] and $\mathbf{u}$ is a three-dimensional velocity vector.

Similalry, ξ_i is a three dimensional velocity vector and is set as follows.

$$\xi_i = \begin{cases} (0,0,0) \text{ for } i = 0, \\ (\pm1,0,0),(0,\pm1,0),(0,0,\pm1), \text{ for } i = 1 \text{ to } 6, \text{ and}, \\ (\pm1,\pm1,0),(\pm1,0,\pm1),(0,\pm1,\pm1), \text{ for } i = 7 \text{ to } 18. \end{cases}$$

In the **LBM** method, the collision and streaming steps are applied for each particle in parallel. Further, the computations are independent across the particles. A work unit in this case can thus be succinctly described as that corresponding to a set of contiguously numbered particles. In this case, the the computation in an iteration numbered $i + 1$ uses the values generated in iteration i, for $i = 1, 2, \cdots$. Apart from this, there is no post-processing involved. Thus, the LBM workload satisfies the characteristics required of our framework from Section III.

A. Implementation

We use the **LBM** D3 Q19 model to simulate fluid flow around a solid sphere. This has a three dimensional fluid lattice and each lattice point has 19 speed vectors as shown in Figure 5. Each thread is allocated to each particle [38], [16] and the 19 directional velocities are stored such as for one direction all points are stored consecutively [41]For a given number of particles, we first apply the collision function which calculates $f(x, t + \delta t)$ and then apply the stream function which calculates $f(x + \delta x, t + \delta t)$. The boundary collisions are treated as elastic collisions. In the stream step for particle numbered n, we need the speed of the particle numbered $n+1$. For that we only need to asynchronously transfer once when GPU and CPU are computing the last iteration in last block.

We use the Bhatnagar-Gross-Krook [5] model for its accuracy. In our implementation, we consider up to 1.5 million particles in three dimensions, and simulate their flow around a solid sphere. The size of the work unit is determined empirically. The CPU works on particles numbered $1, 2, \cdots$, and GPU works on particles with numbers $n, n-1, \cdots$,. The work units are prepared accordingly. The LBM computation is iterative in nature. Since the computation in each iteration is identical, we use the framework from Section III for the first iteration. This allows us to identify the proportion of work to be allotted to the CPU and the GPU respectively. This proportion is used in all the subsequent iterations. This lets us save the overhead of our framework over multiple iterations.

B. Results

In our experiments we use three dimensional particles with coordinates generated uniformly at random. The number of particles is varied from 250 K to 1.5 M. We simulate flow of these points around a solid sphere for 100 iterations for each set of points.

The baseline algorithm in our case is a heterogeneous algorithm adapted from [16] and identifies the proportion of work to be allotted to the CPU and the GPU empirically. To see the benefit of our framework from Section III, we also measured the absolute difference in the work split percentage allotted to the CPU and the GPU in our algorithm and that of the baseline algorithm. The results of this comparison on two heterogeneous computing platforms described in II is shown in Figure 6(a). The absolute difference is seen to be quite small on average.

We also compared the absolute difference in the runtime of our algorithm to that of the baseline algorithm. The results of this experiment are shown in Figure 6(b). As can be seen from Figure 6(b), the absolute difference is under 3% on average on both the platforms. Also, our implementation has a low idle time of 1.5% on both platforms on average. Since we use the framework in the first iteration, and then use the work split percentage as obtained from our framework in later iterations, the difference in the runtime between our implementation and the baseline implementation is rather small.

VII. Conclusions

In this paper, we have proposed and demonstrated a simple and efficient mechanim for dynamic load balancing of heterogeneous algorithms. We also showed the efficacy of our framework on three different workloads. Our results show that our implementations achieve good performance with respect to the work split percentage and runtime compared to corresponding best reported baseline implementations.

Our work therefore seems to partly solve the important problem of engineering heterogeneous algorithms using the work partitioning strategy. The nature of our results indicate that employing our framework, one may sacrifice only a small portion of the runtime but can achieve very good load balancing across heterogeneous devices.

We also identified characteristics of workloads that can benefit from our mechanism. In future, we would like to study our mechanism on other emerging heterogeneous computing platforms. Similarly, it would be interesting to apply our framework to other applications which possess the required characteristics. A final question that is left unanswered in our work is to find anayltical ways to identify the size of the work unit for a given device. However, simple and accurate analytical models are difficult to obtain.

References

[1] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., Yelick, K. A., Demmel, M. J., Plishker, W., Shalf, J., Williams, S., and Yelick, K. The landscape of parallel computing research: A view from berkeley. Tech. rep., TECHNICAL REPORT, UC BERKELEY, 2006.

[2] Bader, D. A., Agarwal, V., and Madduri, K. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IEEE IPDPS* (2007), pp. 1–10.

[3] Banerjee, D. S., and Kothapalli, K. Hybrid algorithms for list ranking and graph connected components. In *in Proc. IEEE HIPC* (2011), pp. 1–10.

[4] Banerjee, D. S., Sakurikar, P., and Kothapalli, K. Fast, scalable parallel comparison sort on hybrid multicore architectures. In *Proceedings of the third International workshop on Accelerators and Hybrid Exascale Systems (AsHES)* (2013).

[5] Bhatnagar, P. L., Gross, E. P., and Krook, M. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical Review 94*, 3 (1954), 511525.

[6] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[7] Buluc, A., and Gilbert, J. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on* (2008), pp. 503–510.

[8] Center, B. S. "SMP superscalar user's manual v.2.4". `www.bsc.es/media/4783.pdf`.

[9] Chen, L., Hu, Z., Lin, J., and Gao, G. R. Optimizing the fast fourier transform on a multi-core architecture. In *Proc. of IEEE IPDPS* (2007), pp. 1–8.

[10] Chen, S., and Doolen, G. D. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics 30*, 1 (1998), 329–364.

[11] Chhugani, J., Satish, N., Kim, C., Sewall, J., and Dubey, P. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS* (2012), pp. 378–389.

[12] Demmel, J. W., and Y, M. T. H. Applied numerical linear algebra. In *Society for Industrial and Applied Mathematics* (1997), SIAM.

[13] Dongarra, J. Personal communication, 2013.

[14] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5 (May 1998), 212–223.

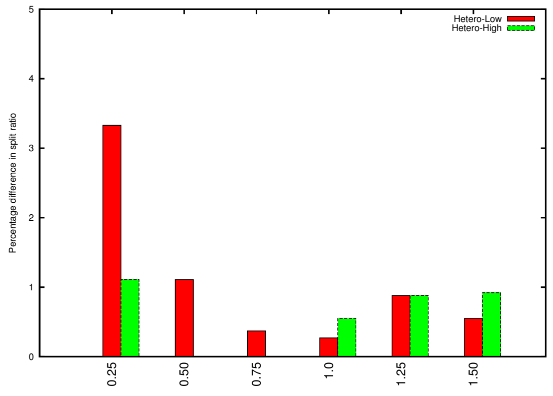
[15] Gharaibeh, A., Costa, L. B., Santos-Neto, E., and Ripeanu, M. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *in Proc. of IEEE IPDPS* (2013).

[16] Habich, J., Zeiser, T., Hager, G., and Wellein, G. Performance analysis and optimization strategies for a d3q19 lattice boltzmann kernel on nvidia gpus using cuda. *Adv. Eng. Softw. 42*, 5 (May 2011), 266–272.

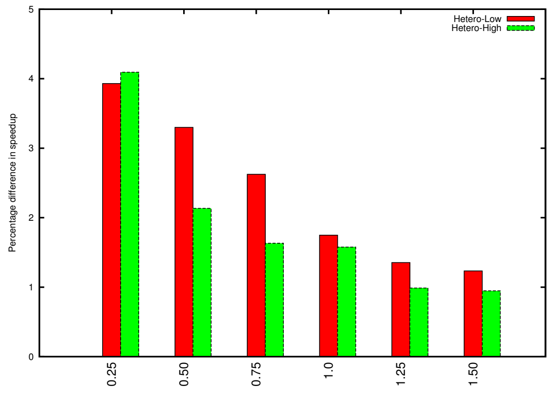
[17] He, L., and Ioerger, T. R. Forming resource-sharing coalitions: a distributed resource allocation mechanism for self-interested agents in computational grids. In *Proceedings of the 2005 ACM symposium on Applied computing* (2005), pp. 84–91.

[18] He, X., and Luo, L.-S. Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation. *Phys. Rev. E 56* (Dec 1997), 6811–6817.

[19] Hong, S., Oguntebi, T., and Olukotun, K. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proc. of IEEE PACT* (2011), pp. 78–88.



(a) Split %



(b) Time

Fig. 6. Figure shows the absolute difference in the work split percentage and runtime with respect to the baseline implementation for **LBM**.

[20] HORTON, G. A multilevel diffusion method for dynamic load balancing. *Parallel Comput.*, 19 (1993), 209 – 218.

[21] INTEL. Math kernel library. `http://developer.intel.com/software/products/mkl/`.

[22] JAJA, J. *An Introduction To Parallel Algorithms*. Addison-Wesley, 2004.

[23] KURZAK, J., BUTTARI, A., AND DONGARRA, J. Solving systems of linear equations on the cell processor using cholesky factorization. *IEEE Trans. Parallel Distrib. Syst. 19*, 9 (Sept. 2008), 1175–1186.

[24] LAN, Z., TAYLOR, V. E., AND BRYAN, G. A novel dynamic load balancing scheme for parallel systems. *J. Parallel Distrib. Comput. 62*, 1763–1781.

[25] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture* (2010), ISCA '10, ACM.

[26] LRIG, C., GROSSO, R., AND ERTL, T. Implicit adaptive volume ray-casting. In *In GraphiCon 97* (1997), pp. 114–120.

[27] MATAM, K. K., INDARAPU, S. R. K. B., AND KOTHAPALLI, K. Sparse matrix-matrix multiplication on modern architectures. In *HiPC* (2012), pp. 1–10.

[28] MATAM, K. K., AND KOTHAPALLI, K. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011), ICPP '11.

[29] MONAKOV, A., AND AVETISYAN, A. Implementing blocked sparse matrix-vector multiplication on nvidia gpus. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pp. 289–297.

[30] MONAKOV, A., LOKHMOTOV, A., AND AVETISYAN, A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proc. of HiPEAC* (2010), pp. 111–125.

[31] NVIDIA. *CUDA CUSPARSE Library*, Aug. 2010.

[32] OF TENNESSEE, U. "PLASMA". `http://icl.cs.utk.edu/plasma/`.

[33] RANDALL, K. H. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[34] SATISH, N., HARRIS, M., AND GARLAND, M. Designing efcient sorting algorithms for manycore gpus. In *Proc. of IEEE IPDPS* (2009).

[35] SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. Scan Primitives for GPU Computing. In *Proc. ACM Symp. Graphics Hardware* (2007), pp. 97–106.

[36] SONG, F., YARKHAN, A., AND DONGARRA, J. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 19:1–19:11.

[37] SUCCI, S. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation)*. Numerical mathematics and scientific computation. Oxford University Press, USA, Aug. 2001.

[38] TOLKE, J., AND KRAFCZYK, M. Teraflop computing on a desktop pc with gpus for 3d cfd. *Int. J. Comput. Fluid Dyn. 22*, 7 (Aug. 2008), 443–456.

[39] TOMOV, S., DONGARRA, J., AND BABOULIN, M. Towards dense liner algebra for hybrid gpu accelerated manycore systems. *Parallel Computing 12* (Dec. 2009), 10–16.

[40] VAN DER HOEF, M., YE, M., VAN SINT ANNALAND, M., ANDREWS, A., SUNDARESAN, S., AND KUIPERS, J. Multiscale modeling of gas-fluidized beds. *Advances in chemical engineering 31* (2006), 65–149.

[41] WELLEIN, G., ZEISER, T., HAGER, G., AND DONATH, S. On the single processor performance of simple Lattice Boltzmann kernels. *Comput. Fluids 35*, 8-9 (Sept. 2006), 910–919.

[42] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. of SC'07*, ACM, pp. 38:1–38:12.

[43] YAMAZAKI, I., DONG, T., TOMOV, S., AND DONGARRA, J. Tridiagonalization of a symmetric dense matrix on a gpu cluster. *in the proceedings of the third International workshop on Accelerators and Hybrid Exascale Systems (AsHES)* (may 2013).

[44] YANG, X., PARTHASARATHY, S., AND SADAYAPPAN, P. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proc. VLDB Endow. 4*, 4 (Jan. 2011), 231–242.

[45] ZWICK, U. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM 49* (2000), 2002.