Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 153–163. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

AN ASYNCHRONOUS API FOR NUMERICAL LINEAR ALGEBRA

ANDRÉ RIGLAND BRODTKORB*

Abstract. We present a task-parallel asynchronous API for numerical linear algebra that utilizes multiple CPUs, multiple GPUs, or a combination of both. Furthermore, we present a wrapper of this interface for use in MATLAB. Our API imposes only small overheads, scales perfectly to two processor cores, and shows even better performance when utilizing computational resources on the GPU.

Key words: asynchronous, multicore, GPU, MATLAB, CUBLAS, double precision

1. Introduction.

Algorithms from numerical linear algebra are important tools with a variety of uses. Common to many of these algorithms is that they require a substantial amount of computation time. Therefore, there has been a lot of research into developing optimized APIs and libraries such as BLAS [14], FLAME [15], LAPACK [2] and PLASMA [13]. However, when these libraries are used, they stall the program execution until each algorithm has completed. This prevents overlapping heavy computation with other operations, such as reading in new data from disk, without complex multithreaded programming. We present an interface that offers asynchronous and task-parallel execution of BLAS functions on different *backends*. This enables easy overlap with other operations and enables us to utilize several processor cores, multiple graphics processing units (GPUs), or both in combination.

The interface we present consists of a frontend that exposes familiar BLAS functions and several backends that implement them. The frontend schedules execution of the algorithms to the different backends run-time based on a set of simple criteria to utilize all available processing power. We further present a wrapper of this interface for use in MATLAB. This is a natural extension to our previous work [8], where we showed speed-ups using the GPU as a computational resource in MATLAB.

We have focused on utilizing multicore CPUs in conjunction with one or more GPUs. However, our ideas could also have been applied to other accelerator cores such as the Cell broadband engine (Cell BE) [16] or field programmable gate arrays (FPGAs) [10].

The rest of this article is sectioned as follows: Section 2 discusses current multi-core hardware and software trends, and we relate our API to existing programming languages and APIs. Readers familiar with these trends might opt to jump straight into Section 3, where we contrast our contribution to related work. Section 4 describes our interface, followed by Section 5 where we show performance results. We end this article with some concluding remarks and future research directions in Section 6.

2. Current Trends in Parallel Commodity Hardware.

The achieved performance of computer programs has traditionally, with only minor adjustments, increased with new hardware generations. The performance gain has mainly come from an increase in four metrics: size of system memory, processor core clock frequency, speed of system memory, and number of instructions per clock cycle. The size of system memory is limited to 4 GiB on 32 bit systems, but this limitation was increased to a theoretical 256 GiB for 64 bit systems with the emergence of the x86-64 instruction set in 2003. The increase of the three other metrics, however, has come to a halt. The factors limiting further growth have been called the *power wall*, *memory wall*, and *ILP wall*, constituting a "brick wall for serial performance" [3].

The power wall prevents further increases in clock frequency. Increasing the clock frequency requires reducing the gate size and increasing the supply voltage. Both reducing the gate size and increasing the supply voltage leads to an increase in leakage power and generated heat. This generated heat is proportional to the square of the frequency, and we seem to have reached a physical limit to what the chips can withstand without exotic cooling.

The size of on-chip caches has increased drastically in recent years. This is because the bandwidth to off-chip memory has become relatively slower and slower, referred to as the *von Neumann bottleneck*. One of the reasons for this slow-down is that there is a physical limit to the number of pins connecting the processor to the motherboard and main memory. The speed per pin is also limited, further obstructing speed increases. This limitation is commonly known as the memory wall.

*SINTEF, Dept. Appl. Math., P.O. Box 124, Blindern, N-0314 Oslo, Norway.

The third wall is referred to as the ILP wall. Increasing the number of instructions executed per clock cycle is a result of instruction level parallelism (ILP). Complex logic executes dependent instructions concurrently by predicting the program flow. Further development of ILP, however, is currently unfeasible because of the exponentially difficult task of predicting future instructions.

These three walls mark the end of serial performance increase for the time being, and the era of multicore computing has begun. Dual- and quad-core processors have already become the mainstream of processors, and we will see a steady increase in the number of cores in the future. This represents a great challenge to computer scientists and algorithm designers as most algorithms are designed for serial architectures, thus unable to benefit from these processor designs.

In addition to the processing power offered by the multicore CPU, most modern computers are also equipped with one or more dedicated graphics cards to accelerate rendering in games. The graphics card looks a lot like a separate computer by itself: it contains a GPU, dedicated graphics memory, and a graphics BIOS. The cards are accessed through the graphics driver that offers one or more graphics APIs.

In current games, you typically have a screen resolution of 1600×1200 pixels, with at least 60 frames drawn each second. This totals to over one hundred million pixels each second. The number of operations per pixels can be very large, as modern games often use advanced rendering techniques requiring several rendering passes per frame. To meet these demands graphics cards have developed an extreme computational capacity. Current consumer level GPUs can have a theoretical peak performance of over 1 TFLOPS, compared to CPUs that have less than 150 GFLOPS. The GPU also outperforms the CPU when it comes to memory bandwidth. Motherboards give the CPU access to main memory at speeds up to 25 GiB/s, while the typical memory speed of a commodity-level computer is around 13 GiB/s, or even less. For GPUs, however, the memory speeds reach over 140 GiB/s. And with good reason. It takes a lot of bandwidth to process over one hundred million pixels each second.

Using graphics processors to accelerate non-graphical applications is a field that has gained in popularity, see e.g., `www.gpgpu.org` and Owens et. al. [25]. One of the main bottlenecks with such use of the GPU is the relatively slow data bus between GPU and main memory. Heterogeneous processor designs such as the Cell BE and the coming AMD Fusion processors remove this bottleneck by incorporating accelerator cores on the same silicon die as the CPU. It is commonly accepted that using such accelerator cores outperform pure CPU implementations for a variety of processing tasks. See for instance Brodtkorb et. al. [9] for an example of a comparison between multi-core CPUs, the GPU, and the Cell BE.

The supercomputer community is also using accelerator cores to achieve higher performance. Bull have disclosed that they will build a supercomputer with 1080 octa-core Nehalem CPUs in conjunction with 96 NVIDIA Tesla [17] GPUs for Grand Equipement National de Calcul Intensif in France. The RoadRunner project at Los Alamos National Laboratory utilizes 6,948 AMD dual-core CPUs in conjunction with 12,960 Cell BE processors, and is the first machine to achieve over one PFLOPS sustained performance. It is currently the most powerful supercomputer in the world, and also one of the most energy efficient.

2.1. Programming languages and APIs. There are already many different programming languages and APIs for multicore computing, such as POSIX Threads (pthreads) [12], Intel Threading Building Blocks (TBB) [27] and OpenMP [24]. All these libraries use threads to utilize multiple processor cores.

POSIX Threads is a low-level API where the programmer handles each thread explicitly. This level of access offers great flexibility, but at the same time requires a lot from the programmer. Programs written using pthreads are subject to thread issues such as *race-conditions*, *deadlocks*, *starvation*, and *priority failures*, all of which require the programmer to create and maintain complex logic.

TBB is a C++ library that uses threads, but focuses on *tasks*. Given a parallel algorithm, TBB aids in partitioning and scheduling the tasks, thus reducing the risk of thread issues. It also implements task stealing that enables dynamic load balancing. The task stealing is performed runtime by splitting large tasks into smaller tasks, and then redistributing them to idle threads. Starting these tasks is 100 times faster than starting a thread on Windows XP.

OpenMP consists of a set of compiler commands. The programmer outlines the parallel sections of the code using a set of pragmas, guiding the compiler where to perform parallelization. The compiler recognizes these pragmas, and generates code that can execute in parallel on a shared memory machine. OpenMP is used in the OpenMP Multi-Threaded Template Library, which is a parallel implementation of `<numeric>` and `<algorithm>` from the standard template library (STL).

The asynchronous linear algebra API we present here uses Boost::Threads, a portable API for threading that is slightly higher level than POSIX Threads. We offer a task-parallel API, using ideas similar to TBB. However, we do not implement task subdivision and task stealing, and focus only on mathematical operations defined in the BLAS API.

Traditionally, the GPU had to be accessed through a graphics API such as OpenGL [28] or DirectX [20]. Accelerating computations using the GPU thus required that the algorithm could be rewritten in terms of operations on graphical primitives, a tedious and error prone solution for non-graphics use. An increasing demand for non-graphical APIs sparked a lot of research into creating other abstractions, and there are currently two dominant APIs for *stream computing*; NVIDIA CUDA [23] and Brook [11].

CUDA is a data-parallel programming language and paradigm that requires algorithms to be structured into blocks of independent computation. It can execute on recent NVIDIA GPUs, or in emulation mode on the CPU. The CUDA Zone at the NVIDIA website contains an updated list of academic and industrial uses of CUDA. This programming paradigm has also been implemented for multicore CPUs as MCUDA [29].

Brook is an open source API developed at Stanford. The Stanford version has not reached a final release, but has spurious updates. AMD has developed an extension to the API called Brook+ [1] for their GPUs, but the language specification is not fully implemented. The Brook API is used by Folding@home project to simulate protein folding on both GPUs and the Cell BE in the PlayStation 3 gaming console.

PyStream is a python interface to CUDA, CUBLAS [22] and CUFFT [21] that supports seamless data-transfer between CPU and GPU memory space. TechX has stopped developing this open source version, but continues developing GPULib, a library that accelerates mathematical functions using the GPU. GPULib offers bindings to Python, MATLAB and the Interactive Data Language (IDL).

RapidMind [19] is a high-level stream programming API with backends for both multicore CPUs, GPUs, and the Cell BE. The backends are themselves responsible for low-level optimization, and RapidMind has published results where highly optimized RapidMind code outperformed hand-tuned code [26].

Our approach is similar to GPULib, as we also offer transparent data-transfers and asynchronous execution of mathematical functions on the GPU. As with RapidMind, we also support different backends for our interface. RapidMind offers a general programming framework for data-parallel execution, whereas we focus on task-parallel mathematical functions. In addition, we support using multiple backends simultaneously for task-parallel execution.

3. Related Work. For an overview of history and recent developments in linear algebra on the GPU, Owens et. al. [25], and the references therein, provide a good summary. We have previously presented an interface between MATLAB and the GPU [8], where algorithms from numerical linear algebra were accelerated by the GPU. This early work required the algorithms to be written in terms of operations on graphical primitives, as only graphics APIs were available for our NVIDIA GPU. Accelereyes are developing a product called Jacket to accelerate MATLAB using the GPU. They use ideas very similar to our original paper but also offer OpenGL visualization.

Faticia and Jeong [18] have also presented a coupling of MATLAB and the GPU. They used CUFFT to accelerate simulations run from MATLAB, and reported the GPU version to be 14 times faster for a 2D isotropic turbulence problem. GLAME@LAB [6] uses CUBLAS to accelerate FLAME@LAB [7], the MATLAB interface to the FLAME API [15]. They showed speedups over Octave for several different algorithms. The performance of CUBLAS has been explored by Barrachina et. al. [5]. They also presented a hybrid sgemm (**s**ingle-precision **ge**neral **m**atrix **m**ultiply) algorithm that utilized both CPU and GPU computational power by splitting the computation into blocks.

Our contribution offers asynchronous execution of mathematical functions that enable us to overlap computation with other operations. We also offer asynchronous data transfer, and task-parallel execution of BLAS functions, being able to benefit from both multiple CPUs, multiple GPUs, or a mixture of both.

4. Interface. We present an asynchronous interface to algorithms in numerical linear algebra. The interface is divided up into a frontend exposed to the user, and one or more backends. The backends can be implemented for different processors, allowing for a heterogeneous processing platform. This section describes the frontend, the backends, and a MATLAB wrapper of the frontend. The interface is general enough to support new backends, e.g., an FPGA backend, and new frontend wrappers, e.g., a Python wrapper.

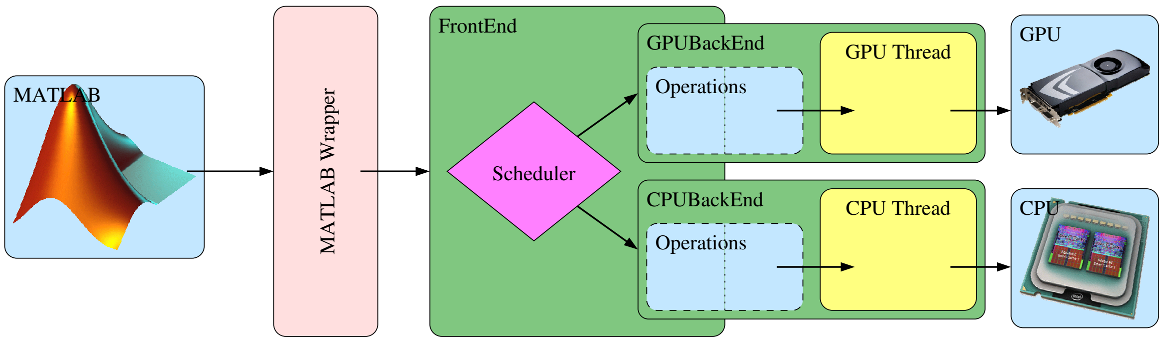

FIG. 4.1. *The interface design.*

Figure 4.1 shows the main layout of the interface. The frontend is responsible for creating *tasks*, and scheduling the tasks to the backends. These tasks are lightweight data-structures that transport pointers to input and output data between the frontend and the backends. The scheduling algorithm tries to find the best backend to schedule the process to, and then adds the task to the queue of the appropriate backend. The backends run asynchronously in separate threads, continuously processing tasks from their task-queue.

4.1. Frontend. The frontend defines functions exposed by our asynchronous API. It is the entry-point for a wrapper or user, and contains a few data-handling functions and a set of functions from the BLAS API. The frontend is flexible with respect to the number, and composition, of backends, where it is up to the user to create and add backends. Once the frontend has one or more backends, it can start scheduling tasks.

Our data-handling functions transfer data between the frontend and the different backends. They create and maintain matrix objects identified by unique IDs. The IDs correspond to matrices previously created by `new_matrix`:

```
MatrixId new_matrix(int rows, int cols, void* data,
   Precision precision, Type type, StorageOrder order,
   Transpose transpose, Copy copy);
```

The first three arguments to `new_matrix` are the matrix dimensions and a pointer to the matrix data. Data allocation is left to the user. The next four arguments are BLAS specific; the data precision, the matrix type, the data storage order, and whether the matrix is transpose. The storage order is assumed to be column-major as currently required by CUBLAS. The final argument is whether the data should be copied to backend specific storage. Setting that the backend does not need to copy the matrix is used when the matrix is an output argument. This prevents the GPU backend from copying data that will be overwritten to the graphics card. The call to `new_matrix` creates a task, but does not schedule it; the task is only scheduled to a specific backend when it is a dependency of another task. The specific details for scheduling these tasks are outlined towards the end of this section.

As the tasks are executed asynchronously, and the user is unaware what backend performs the actual computation, the function `get_matrix` is supplied:

```
void get_matrix(MatrixId id, bool synchronous);
```

The arguments to `get_matrix` are the matrix ID and a *blocking* flag. The `get_matrix` function creates a task to retrieve the matrix from the backend, and the task is immediately scheduled to the backend that holds ownership of the matrix. If the blocking flag is set, the frontend waits until the task has been executed. The CPU backend simply ignores these tasks, as all its data already resides in CPU memory space. The GPU, however, must read the data back from the GPU to CPU memory if the GPU data has been altered. Using `get_matrix` without the blocking flag enables asynchronous read-back of data.

The last data-handling function offered by the frontend is `delete_matrix`, which creates a task that deallocates backend specific resources occupied by the matrix:

```
void delete_matrix(MatrixId id);
```

For the CPU backend this function does nothing, while the GPU backend frees GPU memory consumed by the matrix. The user is responsible for deallocation of the data pointer supplied to `new_matrix`.

The three functions described in the previous paragraphs enable the user to create, read and delete matrices with our asynchronous API. We will now describe how computational tasks are created, scheduled and executed using the Level 3 BLAS double precision general matrix multiply (dgemm) algorithm as an example. The CBLAS [14] function `dgemm` is defined as

```
void cblas_dgemm(const enum CBLAS_ORDER Order,
        const enum CBLAS_TRANSPOSE TransA,
        const enum CBLAS_TRANSPOSE TransB,
        const int M, const int N, const int K,
        const double alpha, const double *A, const int lda,
        const double *B, const int ldb,
        const double beta, double *C, const int ldc);
```

which computes

$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C, \quad \mathrm{op}(X) = \{X, X^T\},$$

using the three matrices A, B and C. Our frontend signature is slightly different,

```
void dgemm(int m, int n, int k,
        double alpha, MatrixId aId,
        MatrixId bId,
        double beta, MatrixId cId);
```

but performs the same computation. We have removed the `Order`, `TransA` and `TransB` arguments, as these are incorporated into the `new_matrix` function. We have also altered the way input and output matrices are sent to the function. Instead of sending raw data pointers, we send matrix IDs that correspond to matrix objects.

The implementation of `dgemm` in our frontend is quite simple. We start by looking up the three matrices identified by the matrix IDs. Then we create a `dgemm` task to hold the parameters and invoke `schedule` to schedule the task to a backend.

The `schedule` function has two tasks: first to group the incoming task with its dependencies into a cluster, and second to find the optimal backend to schedule this cluster to. The grouping is trivial, as all tasks have a dependency list. Selecting the optimal backend to schedule the cluster to, however, is nontrivial. We use two criteria to give an estimate of how good a candidate each backend is, and then schedule the cluster to the backend with the best score. The first criterion is based on the load for each backend, and the second on where dependent tasks have been scheduled.

Our load criterion tries to equalize the load of different backends. It consists of three parts: wall time spent computing, an estimate of wall time needed to process the current task queue, and an estimate of wall time needed to compute the incoming cluster. The time needed to process future tasks is estimated by multiplying the average time to compute a single task by the number of tasks in queue. This is a good approximation if the tasks require similar processing time, such as many matrix multiplications of equally sized matrices. There is one problem, however. When we start our frontend, we do not have enough data to calculate a valid average time needed to compute a single task. For a heterogeneous processing environment, the different backends can have orders of magnitude of difference in performance. This is where our automatic tuning comes into play. We have a pre-compilation step that benchmarks each backend to find a static average. We calculate this average by computing several matrix multiplications for a fixed matrix size for each backend. This gives a relationship between the processing power of the different backends. We incorporate this speed estimate into our source code by defining a C preprocessor macro, before recompiling the program. This constant estimate is used when the number of tasks processed by each backend is too low to give a proper run-time estimate.

The second criterion is computed by estimating the time it takes to move existing dependencies from another backend to the current backend. As you typically perform many operations on the same data, you will encounter situations where dependencies reside on different backends. Moving data between CPU backends is relatively inexpensive. Moving data between a CPU and a GPU, however, is very expensive, and should be penalized by this criterion. For each backend, we loop through the dependencies and estimate the time needed to move each dependency. If the dependency already resides on the current backend, the cost is zero. For dependencies on other backends, however, the cost is calculated as the estimated time needed to get the matrix from its current backend and transfer it to the new backend. The time needed to get the matrix requires a synchronous get, and is accordingly very expensive.

When the best backend has been selected based upon the above mentioned criteria, we synchronously get the dependencies that have to be moved. Then we reschedule them to the selected backend together with unscheduled dependencies, followed by the computational task itself. Continuing our `dgemm` example, this means that the `new_matrix` tasks for the matrices `a`, `b`, and `c` will be enqueued to the selected backend just before we enqueue the `dgemm` task.

4.2. Back-end. Each backend has two parts, as indicated by Figure 4.1. It consists of two threads of execution: the frontend thread and the backend thread. The frontend thread is shared by multiple backends, while each backend runs in a distinct thread of execution. Functionality for enqueuing tasks and querying load status is only called from the frontend thread, and functionality for dequeuing and executing tasks is only called from the backend thread. One can view this relationship between the frontend thread and the different backend threads as a typical *boss-worker* relationship. The frontend acts as the boss, distributing tasks among the workers. In this section we use the term *boss* to signify the thread of execution that runs the frontend, and *worker* to signify the thread of execution that runs the backend selected by the scheduler for the current cluster.

When a task is scheduled to a specific worker, the boss simply adds the task to the task-queue of the selected worker. The boss then notifies the worker of a change, triggering the worker to check its status. While the queue is empty, the worker simply idles waiting for this notification. When notified, and a task has been added to the queue, the worker starts dequeuing and processing the tasks. For the dequeued task, it starts by identifying the type (`new_matrix`, `delete_matrix`, `dgemm`, etc.). When the task has been identified, the worker executes the corresponding function, and continues to process the rest of the queue.

In Section 4.1 we explained how the scheduler enqueued a cluster of tasks to a backend based on different criteria. For the dgemm example, the cluster contained the tasks to create the three matrices, `a`, `b`, and `c`, and the `dgemm` task that used these matrices. We continue the example by explaining how the backend processes and executes these tasks in its queue. Assuming that all four tasks have been added to the task-queue of the GPU backend, we start by dequeuing the first. It is the task to create matrix `a`. The GPU backend allocates GPU memory, and transfers the matrix data into the newly allocated memory. We follow the same procedure for matrix `b`, but for matrix `c` we only allocate memory, assuming we have set the copy flag to false when creating the task. We continue by dequeuing the dgemm task. This instructs us to call the CUBLAS dgemm function using the parameters and matrices defined by the task. When we have completed the dgemm task, we check the queue, and, if empty, wait for notification from the frontend.

4.3. MATLAB wrapper. We have implemented a MATLAB wrapper to offer a high-level interface to the asynchronous linear algebra API. The wrapper consists of two parts, one written in C++, and one written in the MATLAB language (M). The C++ part is very lightweight and thin, simply translating a MATLAB call into a call to one of the frontend functions. The M part defines a new class in MATLAB, and uses operator overloading for this class. Internally, this MATLAB class calls the C++ part.

MATLAB can execute user-defined MATLAB executable (MEX) files that are programmed using FORTRAN or C/C++. These MEX files can be called from MATLAB as if they were regular MATLAB functions. When a MEX file is called MATLAB executes the mexFunction, the entry point all MEX files must define:

```
void mexFunction(int nlhs, mxArray* plhs[],
                 int nrhs, const mxArray* prhs[]);
```

The arguments to the mexFunction are the number of left-hand arguments, pointers to the left-hand arguments, number of right-hand arguments, and pointers to the right-hand arguments, respectively. A general MATLAB call can be written as

`[a, b, ..., y, z] = fun(`α`, `β`, ..., `ψ`, `ω`).`

If `fun` is the name of a MEX file, `plhs` would contain the arguments `a` through `z` and `prhs` would contain arguments α through ω. In our wrapper, we use a single MEX file, where the mexFunction calls the correct frontend function based on the first argument. Our MATLAB function calls are on the format

```
matrix_id = async(funk_id, ...),
```

where `async` is the name of our MEX file, `matrix_id` corresponds to the internal ID used by our asynchronous API, and `func_id` determines what function the wrapper should call. The rest of the arguments are sent to the frontend function is specified by `func_id`.

When creating matrices, the C++ part of the wrapper automatically makes the memory allocated by MATLAB *persistent*. This prevents MATLAB for using automatic garbage collection, effectively giving us ownership of the data.

MATLAB has support for classes in its M script language. This enables us to call the MEX file in a more elegant way than explicitly calling `async` as indicated above. The M part of our wrapper consists of a new class, called *Matrix*, that uses operator overloading to offer an API with familiar MATLAB syntax.

Our operator overloading for the Matrix class translates operations and function calls involving a Matrix object into calls to `async`. Matrix multiplication, for example, would be defined in the file `@Matrix/mtimes.m` in the following way:

```
function c = mtimes(a, b)
c.id = async(42, a.id, b.id);
```

Here, 42 is the function id corresponding to dgemm, `a.id` is the id of matrix `a`, and likewise for `b.id`.

The following MATLAB code shows how this is used in practice to compute dgemm:

```
m = 10; n=20; k=30;
a = Matrix(rand(m, k));
b = Matrix(rand(k, n));
c = a*b;
c_data = double(c);
```

First, `a` and `b` are created as two Matrix objects that contain randomly generated data. Then the multiplication is run using operation overloading. Notice that we do not create the matrix `c`. The C++ part of the wrapper handles this by noticing that `c` has an invalid matrix ID, and thus creates a correctly shaped matrix for us. Finally, we convert `c` from a Matrix object back to double-precision data using the overloaded `double` function. This conversion actually performs a synchronous `get_matrix` that will inhibit performance. As our API is asynchronous we also offer a `read` function that corresponds to an asynchronous `get_matrix`. If there is enough time to read back the matrix before the double-precision data is required, we experience very fast data access.

5. Benchmarking and Analysis. The performance of our interface depends on two main factors; the efficiency of the underlying BLAS implementation and the time spent scheduling a task. The efficiency of several BLAS implementations has previously been analyzed and is not discussed here (see e.g., Barrachina et. al. [5] and the NCSA BLAS Performance Comparison). We show how our API behaves with different backend setups, and try to analyze the results. We further benchmark our scheduling algorithm, and the MATLAB wrapper of the API.

We utilize CUBLAS in our GPU backend. CUBLAS is part of the NVIDIA CUDA SDK, and implements single and double-precision BLAS functions using the GPU. It is steadily being developed, and provides a good interface to GPU accelerated BLAS functions. For our CPU back-ends, we use a single-threaded ATLAS implementation.

We have benchmarked our algorithms on two setups. The first setup consists of a 2.4GHz Intel Core 2 Q6600 quad-core processor with eight GiB of 12.8 GiB/s system memory, and an early engineering sample of the next generation NVIDIA Tesla T10 card (not full performance). The second system consists of a 2.4GHz Intel Core 2 E6600 dual-core processor with four GiB of 12.8 GiB/s system RAM, and an NVIDIA GeForce 8800 GTX graphics card (G80).

Our benchmark runs one thousand computations using 1000×1000 matrices, and timing results have been consistent throughout our benchmarking. The following is a pseudo code of the benchmark:

```
start timer;

for 1..10
  create input and output matrices;
  schedule computation on matrices;

for 11..990
  wait for result;
  delete matrices;
  create input and output matrices;
  schedule computation on matrices;
```

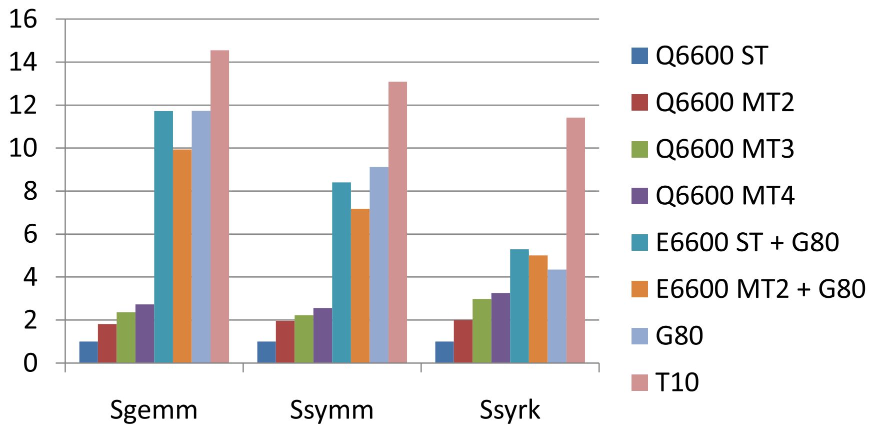

Fig. 5.1. *Speedup results from benchmarking single-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).*

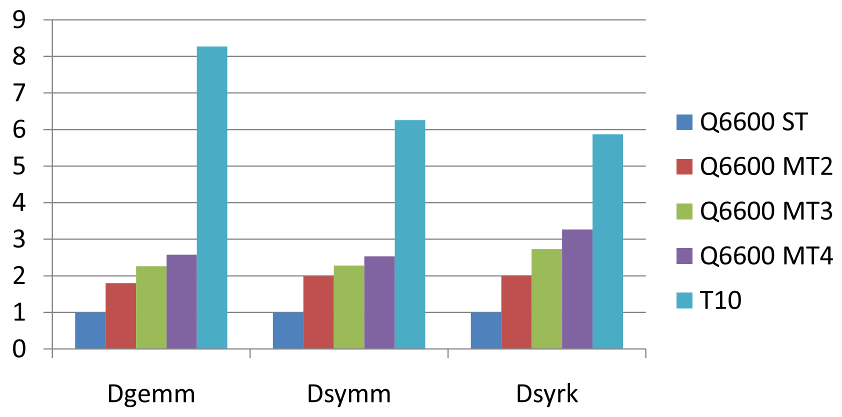

Fig. 5.2. *Speedup results from benchmarking double-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).*

```
for 991..1000
  wait for result;
  delete matrices;

stop timer;
```

This is the worst case scenario, where we only run one computation per matrix. This severely affects the GPUs performance, as transferring data between CPU and GPU memory is expensive. In real world use, one would typically reuse data in multiple computations.

We have implemented and benchmarked three algorithms from BLAS in both single- and double-precision: general matrix multiply, symmetric matrix multiply, and rank-k symmetric matrix update. We have split the results into a single and double-precision part, as the G80 GPU does not support double-precision. Figure 5.1 shows the speedup of different backend setups compared to using a single CPU backend for single precision computations. Utilizing two CPU cores scales almost perfect for all algorithms. The average speedup is 1.92×. For three cores, however, we see a distinct difference between the algorithms. Ssyrk continues to scale almost perfect, with a speedup of 2.98, while sgemm and ssymm achieve a much smaller speedup. We believe this is

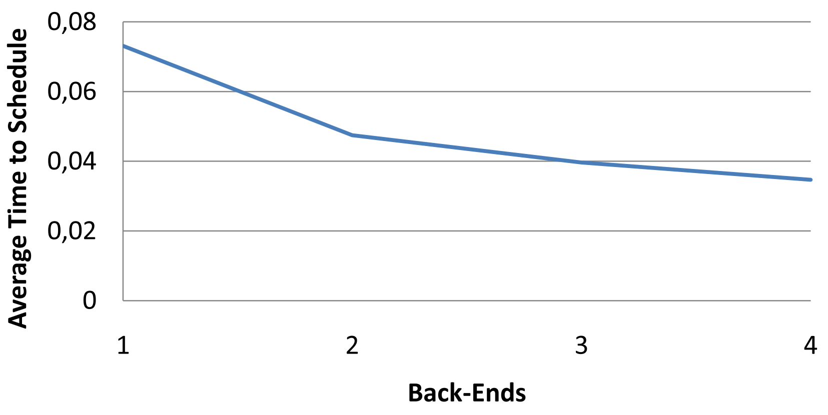

Fig. 5.3. *Average time to schedule a single task compared to the number of backends*

because these two algorithms start trashing the cache. The Q6600 processor has a shared L2 cache of 4MiB per core pair. For two CPU backends, each backend can run on a separate core-pair with exclusive access to the 4MiB cache. For three backends, however, two of the backends must run on the same core-pair, having to share the cache. Ssyrk only uses two thirds of the memory sgemm and ssymm uses, and does not seem to be affected by having to share the processor cache.

When we have four backends running simultaneously, we do not have enough processing power for the scheduler thread to keep all back-ends occupied. Therefore, for four backends, none of the backends scale perfectly, and we do not get the expected speedup.

Figure 5.2 displays the gained speedups for double precision computations, showing similar results to our single-precision equivalents.

Utilizing the GPU backend outperforms using multiple CPU backends by far. This even though we transfer data to and from the GPU for each pass. Utilizing a GPU backend and CPU backend, however, does not scale as well as we could have hoped. We actually see a speed-down compared to using just the GPU for sgemm and ssymm. The reason for this is our scheduling algorithm. By examining the choices made by our scheduler, we find that it makes sub-optimal decisions when faced with a heterogeneous set of backends. We have currently not resolved this issue, but feel confident that we will remove it.

Figure 5.3 shows the average time spent scheduling a task compared to the number of backends. Between 0.04 and 0.08 seconds is a relatively long time. For small matrices this constant scheduling time will affect the experienced performance. Nevertheless, when the tasks we schedule take on the order of seconds to complete, this overhead is negligible. Our API is designed for computation on large matrices, making this overhead less important. However, we believe that optimizing the scheduling algorithm and front- and backend interaction will decrease this overhead.

Our MATLAB wrapper tries to be light-weight and add as little overhead as possible. Our benchmarks show that using our MATLAB wrapper only imposes a constant overhead. Calling our MEX file directly (without the M part of the wrapper) imposes an overhead of 0.3 milliseconds. Using the overloaded functions in the M wrapper is more expensive, as MATLAB must look up the correct function before the calls the C++ part of the wrapper are made. Nevertheless this only imposes an overhead of 2 milliseconds.

There has been skepticism in the scientific community towards using GPUs. It was only recently that GPUs started supporting single-precision arithmetics. These arithmetics, however, do not fully comply with the IEEE standard. Some minor parts such as denormals deviate from the standard, but the floating point implementation has been sufficient in practice for most uses. However, the lack of double-precision numbers has been criticized by the scientific community. Some algorithms require double, or even higher, precision. One example is numerical linear algebra, where even small floating point rounding errors can give large deviations in the results. Using double-precision improves the stability in this respect.

The NVIDIA Tesla T10 GPU supports double-precision in hardware. We do not give a thorough analysis of the double-precision implementation, but offer our early experiences with it. Figure 5 shows the absolute

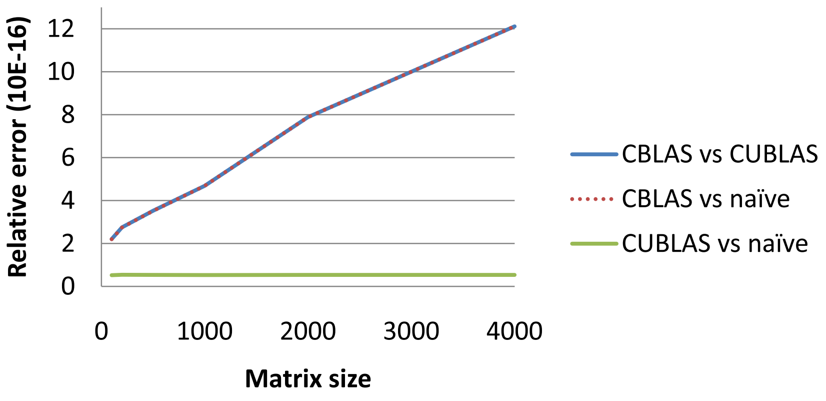

Fig. 5.4. *Measured relative difference per matrix element between CBLAS, CUBLAS and a naïve implementation of double-precision matrix multiplication.*

difference between CBLAS, CUBLAS, and a naïve (double for-loop) CPU implementation of double-precision general matrix multiply (dgemm). There seems to be a linear correlation between the matrix size and the difference between CUBLAS and CBLAS results. To put this into perspective, we also show the difference between CUBLAS and our naïve dgemm implementation. The difference per element between CUBLAS and our dgemm is constant at around $5 \cdot 10^{-17}$. This suggests that the CBLAS implementation reorders operations (e.g., using *Strassens algorithm*) giving different rounding errors than CUBLAS and the naïve algorithm. Reordering operations on the GPU in the same manner (if possible) should thus lessen the experienced difference between CUBLAS and CBLAS.

We will experience that results change between different runs using our API. This is because we do not know a priori where the results will be computed, and the fact that there is a small difference between CBLAS and CUBLAS results. For many uses, this is irrelevant, but for regression testing, for example, one must be aware of the possibility of changes in the result.

6. Conclusions and Future Work. We have presented a task-parallel asynchronous API for numerical linear algebra that imposes only small overheads. This API scales almost perfect to two CPU cores, and is capable of utilizing GPU processing resources for even higher performance. We have also demonstrated that the API can be used through high-level languages, such as MATLAB.

Our scheduling algorithm shows a proof-of-concept that works well for a homogeneous processing environment, but it is suboptimal for a heterogeneous composition of backends. It will be an interesting task to alter the scheduling algorithm to better utilize such heterogeneous processing platforms.

7. Acknowledgements. We would like to thank NVIDIA for the early engineering sample of the next generation Tesla T10 GPU.

REFERENCES

[1] AMD Corporation, *Brook+ language specification, version 1.0 beta*, May 2008.
[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide (third edition)*, Society for Industrial and Applied Mathematics, 1999.
[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The landscape of parallel computing research: A view from Berkeley*, tech. report, EECS Department, University of California, Berkeley, 2006.
[4] M. Baboulin, J. Dongarra, and S. Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures.*, tech. report, 2008.
[5] S. Barrachina, M. Castillo, and E. S. Q.-O. Francisco D. Igua and, R. Mayo, *Evaluation and tuning of the level 3 CUBLAS for graphics processors*, in Workshop on Parallel and Distributed Scientific and Engineering Computing, 2008.
[6] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, *GLAME@lab: An M-script API for linear algebra operations on graphics processors*, 2008.

[7] P. Bientinesi and E. S. Q.-O. et al., *FLAME@lab: A farewell to indices*, tech. report, The University of Texas at Austin, Department of Computer Sciences, 2003. Draft.
[8] A. R. Brodtkorb, *The graphics processor as a mathematical coprocessor in MATLAB*, in CISIS 2008 The Second International Conference on Complex, Intelligent and Software Intensive Systems, 2008.
[9] A. R. Brodtkorb and T. R. Hagen, *A comparison of three commodity-level parallel architectures: Multi-core CPU, the Cell BE and the GPU*. Submitted to PARA'08 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, May 2008.
[10] S. Brown and J. Rose, *Architecture of FPGAs and CPLDs: A tutorial*, IEEE Design and Test of Computers, 13 (1996), pp. 42–57.
[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: stream computing on graphics hardware*, in SIGGRAPH '04, ACM Press, 2004, pp. 777–786.
[12] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
[13] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures (lapack working note 191)*, tech. report, Innovative Computing Laboratory, 2008. `http://icl.cs.utk.edu/plasma/`
[14] J. Dongarra, *Basic Linear Algebra Subprograms Technical forum standard*, International Journal of High Performance Applications and Supercomputing, 16 (2002), pp. 1–111.
[15] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *FLAME: Formal Linear Algebra Methods Environment*, ACM Transactions on Mathematical Software, 27 (2001), pp. 422–455.
[16] IBM, Sony, and Toshiba, *Cell Broadband Engine programming handbook version 1.1*, April 2007.
[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, *NVIDIA Tesla: A unified graphics and computing architecture*, IEEE Micro, 28 (2008), pp. 39–55.
[18] W.-K. J. Massimiliano Fatica, *Accelerating MATLAB with CUDA*, 2007.
[19] M. D. McCool, *Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform*, November 2006. GSPx Multicore Applications Conference.
[20] Microsoft corporation, *MSDN DirectX developer center.*
[21] NVIDIA corporation, *CUDA CUBLAS library version 1.1*, September 2007.
[22] ———, *CUDA CUFFT library version 1.1*, October 2007.
[23] ———, *NVIDIA CUDA programming guide version 1.1*, November 2007.
[24] OpenMP Architecture Review Board, *OpenMP application program interface version 3.0*, May 2008.
[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, *GPU computing*, Proceedings of the IEEE, 96 (2008), pp. 879–899.
[26] RapidMind, *Cell BE porting and tuning with RapidMind: A case study.* Online. `http://www.rapidmind.net/pdfs/RapidMindCellPorting.pdf`
[27] J. Reinders, *Intel Threading Building Blocks:Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Inc, 2007.
[28] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, Addison-Wesley, sixth ed., 2007.
[29] J. Stratton, S. Stone, and W. mei Hwu, *MCUDA: An efficient implementation of CUDA kernels on multi-cores*, tech. report, University of Illinois at Urbana-Champaign, 2008.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008