OpenCLunk: a Cilk-like framework for the GPGPU

Markus Peloquin et al.
markus@cs.wisc.edu

December 20, 2010

Abstract

The potential for GPUs to attain high speedup over traditional multicore systems has made them a popular target for parallel programmers. However, they are notoriously difficult to program and debug. Meanwhile, there are several programming models in the multicore domain which are designed to be simple to design and program for, such as Cilk. OpenCLunk attempts to combine Cilk's ease of programming with the good performance of the GPU. We implemented a Cilk-like task-parallel programming framework for the GPU, and evaluated its performance. The results indicate that while it is possible to implement such a system, it is not possible to attain good performance, and therefore a Cilk-like system is currently not feasible as a programming model for the GPU.

1 Introduction

Graphical Processing Units (GPUs) are becoming more and more common, and have recently started being used for purposes outside of graphics acceleration. They can enable some programs to attain large speedups; speedups of greater than 100 are frequently reported, and supercomputers now commonly make use of them. However, they are also much harder to program for than traditional multiprocessors, and debugging is difficult. One of the main reasons for this is that they are a much more recent innovation than CPUs, and as such, there has not been time for as many helpful tools to be developed. Another reason is that they are more limited than CPUs in what they support, so programmers commonly have to put work into changing their algorithms to work on them (for example, rewriting recursion-based code to use stacks instead).

In contrast, there are a number of programming models, tools, and languages that exist to exploit parallelism on CPUs. One example is the task-based parallel language Cilk. It uses a simple but powerful extension of C to allow programmers to easily write code.

We decided to try to combine the elegance of Cilk with the acceleration of the GPU. From the start, we knew that this would not be simple. Cilk is designed to be a task-parallel language, and GPUs excel at data parallelism. However, we thought that it might be advantageous to have a model that was simple to program, even if the speedup was more modest than the exceptional cases often cited. Therefore, we had several goals. The first was to simply implement a Cilk-like model, where Cilk programs could be easily ported and run: something simple enough for a compiler to target. Our second goal was to use some benchmarks to evaluate the performance of this model, and to prove that it is possible to run programs in a task-parallel manner upon a GPU. Finally, we wanted to characterize the performance of our benchmarks and determine the feasibility of using our implementation as a tool to attain speedup on a GPU.

There has been other work on adapting existing CPU parallel programming platforms onto GPGPUs. An OpenMP to CUDA compiler extension called O_2G was developed with generally positive results [4]. Since both platforms are already data-parallel, the authors focused mostly on modifying code to fit the GPGPU's memory model. Because it is already fairly well understood how to write data-parallel code for the GPU, and because there are already tools to aid in development and debugging, we wanted to focus on another model, and a task-parallel model has not been widely studied for the GPU. Hence, we chose to implement a Cilk-like model.

Section 2 gives a background of the technologies involved. Section 3 describes the details of our imple-

mentation's design and interface. Evaluation is covered in Section 4. We discuss the OpenCLunk model in 5 and conclude in Section 6.

2 Background

2.1 GPGPUs

As the difficulty of improving uniprocessor performance has increased, the landscape has changed to include other architectures besides CPUs. GPUs have been used to accelerate graphics calculations for some time, but recently they have been repurposed for general-purpose computation, under the name GPGPUs. Calculations are offloaded from the CPU onto the GPU and executed there.

GPGPUs differ from CPUs in a number of ways. They are intended for data-level parallelism: many threads all executing the same instructions at the same time. They have far more cores than typical multicore systems; they also can run hundreds of threads simultaneously. Transcendental functions are implemented in hardware, allowing them to be very fast, especially for single-precision floating point operations.

However, there are some disadvantages of GPUs as opposed to CPUs. Threads are executed in thread groups, typically of 32 threads. Within these groups, all threads execute the same instructions. If there is an `if` condition that only evaluates true for some threads, a form of predicated execution will disable the threads for which the `if` condition was false, then perform the instructions in the `if` block. Afterwards, those threads may be disabled and the others enabled while the instructions in an `else` block are executed. This is called divergence, and limits performance. Another limitation is in what types of code can be run on the GPU. One limitation that is particularly salient to our task was that recursive calls are not possible since their use of memory is unbounded.

2.2 OpenCL

OpenCL (Open Computing Language) is a C-based language for writing kernels, which can then be executed on GPUs and CPUs [2]. It is portable between different manufacturers: it can be used for both ATI's and NVIDIA's devices. That was a main motivation in choosing OpenCL over CUDA. OpenCL claims to support task-level parallelism in the form of command queues and the option `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` [3]. However, this is not supported on most architectures. Unfortunately, this is not so much an instruction to OpenCL as it is a suggestion. No errors or warnings are given when this option is used for hardware that does not support it. Since the hardware we had access to does not support out-of-order execution, we implemented a framework using other features, which we will describe later.

2.3 Cilk

Cilk is an extension of the C programming language to enable easy parallelization of programs [1]. It uses a task-based parallelism model. If the Cilk keywords are removed from a Cilk program (the C elision is taken), the program executes correctly in serial, simplifying program design and debugging. The main Cilk keywords are `spawn` and `sync`. The `spawn` keyword is used to indicate that a function call can run in parallel with the main method and other spawned functions. The `sync` keyword pauses the execution of a thread until all of previous spawns in the same function call have completed. It is common for Cilk programs to be recursive, with each function having multiple recursive calls to itself.

The way that these keywords are implemented in Cilk is to use work-stealing queues. When a thread spawns a new function, it adds it to the head of its work queue. Other threads can steal from the tail of other queues when they exhaust the work in their own queue. When a thread steals, it converts the function to a slower version that executes synchronization statements; in the common case, synchronization is unnecessary because by the time the thread reaches a `sync` statement, it would have executed all previous instructions anyway. This allows for very cheap spawns, which is part of why Cilk has very good performance.

We chose to use Cilk as the model for our system because it is an easy to use programming model, yet powerful enough to implement a wide variety of programs. Because our platform was different, we did not however use the same design for our implementation; in particular, we made no use of work-stealing queues.

3 OpenCLunk

OpenCLunk is our implementation of a task-parallel model for GPGPUs. Its name is a combination of OpenCL, Cilk, and 'clunk', and serves to describe both its roots and its performance.

3.1 Execution model

The way tasks execute in OpenCLunk is similar to what happens in Cilk, but necessarily differs in some key ways. The first and most important problem to solve was how to spawn tasks. As the GPGPU environment is quite restrictive, our solution was for a task to return whatever arguments would have been used in the spawn invocations back to the CPU. These spawn arguments then serve as input arguments for subsequent spawned tasks. The only other thing a task might respond with is a return value, corresponding with the base case of a recursive function. As a simplification, we force the number of spawns to be constant across all tasks.

Tasks are not actually spawned until the parent tasks return to the CPU. The *sync* step from Cilk can therefore only be implicit. We also needed to support code that would run after the spawned tasks complete. This might include summing the results of the spawned tasks or finding the maximum. We decided for simplicity to have the post-sync code, the *reduction*, execute on the CPU. This is acceptable since reductions rarely do intensive work, and the additional migration costs for this simple action would be prohibitively large.

3.2 Task groups

Tasks are executed in a series of task groups. The first task group is merely the root task. It will be sent the initial arguments that the user specified, but it is otherwise indistinguishable from the other task groups. Each subsequent task group is composed exclusively of tasks that were spawned off of the previous group. Since tasks all take the same amount of time to execute, each level of the execution tree can be invoked at once, reducing the communication traffic with the GPGPU.

It would have been convenient to use the spawn arguments of one task group as the arguments for the subsequent task group. One benefit of this approach would have been less memory bandwidth: once a task group completes, they need only transfer back some relatively small status information. This has the problem of leaving holes in the arguments, since not all tasks will spawn. As subsequent levels of spawns execute, those holes take up more and more memory. Our approach was to copy the spawn arguments back to the CPU. Task groups would then be rearranged so that all spawning tasks would be at the front, moving the remainder at the back.

One of our previous approaches was to have each group consisting solely of the spawned tasks of the previous group. This offered a more fluid execution, but it had higher memory requirements for the task groups and additional communication with the GPGPU. It also depended on the OpenCL device supporting out-of-order execution, which our hardware did not support. It did, however, simplify the reduction process and did not require rearranging the task groups, so it had some computational advantage.

3.3 Dependencies

All tasks have a result value, but it is often dependent on the results of the tasks the parent has spawned. Whenever a task group completes execution, OpenCLunk scans through the completed tasks in this group. For each completed task, the value is copied into a buffer belonging to its parent. If this was the last task to add its result to its parent, the reduction operation is performed, and the parent takes on that result. This process repeats when the parent in turn copies its new result to its parent.

3.4 Execution process

When a user runs an OpenCLunk function, a loop begins that terminates when the root gets a value. Before the loop, a root task is created and invoked on the GPGPU. On completion, OpenCL asynchronously will add completed task groups to a queue in OpenCLunk using event callback functions. Each completed task group is handled within the loop. First, a task group is removed from the queue. Its tasks are rearranged before making a new child task group, which is subsequently sent to OpenCL for execution. Once they are added to OpenCL's command

queue, dependencies are handled and reductions performed. The last task group will be able to resolve all remaining dependencies, and OpenCLunk returns the final value.

3.5 Restrictions

Tasks may only choose to spawn or to not spawn, so we require that a task always spawn a constant number of times. This is sufficient for many benchmarks, and simplifies the design. It would be possible to allow tasks to spawn up to N times by returning a bitmask value $\lceil N/32 \rceil$ bits long. This would have caused the code in the user-level kernel functions, the code performing the task group rearranging, and the code performing reductions to become more complicated.

As it is currently, OpenCLunk can only handle problem sizes up to some threshold. Task groups require a particularly large buffer for their spawn arguments; for recursive Fibonacci, it doubles in size nearly every time. Resources can be exhausted, and so global work sizes have per-algorithm limits. The way to get around this would be to divide the tasks into two task groups. The two task groups would need to be executed in serial, and task group sizes would never decrease. The consequence of this situation is that throughput would become constant. For the purpose of benchmarking when resources are exhausted, it improves little. The average speedup could increase, but only up to some limit. For simplicity then, we do not implement this capability.

There are also some limitations in what features OpenCLunk supports. OpenCLunk is similar to Cilk, but does not implement all of its features. For example, Cilk includes an explicit `abort` keyword to allow all tasks to be stopped. This is useful in cases like searches where only the first result needs to be returned, and execution can stop at that point. It is currently possible to emulate `abort` functionality in OpenCLunk by means of a read-write buffer parameter that is checked at the beginning of each kernel, and if it is found to be set to a value indicating an abort, the task returns immediately without doing any work or spawning any children. However, it would be possible to implement a better `abort` that stopped the CPU portion of the code from starting more task groups.

Another limitation is that tasks can only make recursive spawns. In Cilk, any function can be spawned from within any other function. This feature could be implemented by passing back the name of the function to be called (or an enumerated representation of it), but it is not currently available in OpenCLunk.

Finally, although many Cilk functions can be rewritten so that all `sync`s take place at the end of the kernel and results are combined in a reduction phase, this is non-trivial or even impossible for some cases. For example, in some Cilk programs, there is a phase with several `spawn`s followed by a `sync`, followed by additional `spawn`s. Alternately, the end result of a Cilk function has a more complicated operation than addition performed on it, such as a multiplication by some factor at each step. These patterns of execution are not allowed in OpenCLunk.

3.6 Writing for OpenCLunk

Ideally, the user would be able to use a few simple keywords, and the compiler would generate the code needed to run on the GPU. However, since this project was primarily a proof-of-concept, the initial implementation of OpenCLunk requires the user to make use of a somewhat clunky interface. The intention was to determine whether OpenCLunk would be successful before worrying about the value of having a compiler and attractive interface.

There are three primary components to an OpenCLunk algorithm implementation. Most of the recursive function survives as the kernel function. The user must also specify the arguments in the CPU code. Reductions are separate from the kernel and are a part of the CPU code as well. This subsection gives example code for the recursive Fibonacci algorithm, but a more general and complete example can be found in Appendix A.

3.6.1 Kernel function

Most important is the kernel function, which is the majority of what in Cilk would be recursive function. It has some base case that returns a value, as well as necessary code to spawn child tasks. The recursive kernel of the recursive Fibonacci algorithm is shown in Listing 1. The first argument, `max_tid` is the global work size. `arg_n` is the list of arguments sent to the tasks, indexed by `tid`, and `spawn_n` is the list of output arguments, indexed both by `tid` and the spawn

offset. Though not strictly necessary, the list of arguments should be declared with `__constant` to take advantage of caching; the rest are writable and must be `__global`. Lines 11–15 are the base case; the task indicates it should not spawn and its result is `n`. The remainder sets up the spawn arguments: the task indicates it would like to spawn and specifies what to use for the arguments to the spawned tasks.

```
__kernel void fib(
    unsigned max_tid,
    __constant unsigned *arg_n,
    __global unsigned *result,
    __global unsigned *spawn,
    __global unsigned *spawn_n)
{
    unsigned tid = get_global_id(0);
    if (tid > max_tid) return;
    unsigned n = arg_n[tid];
    if (n < 2) {
        spawn[tid] = false;
        result[tid] = n;
        return;
    }
    spawn[tid] = true;
    spawn_n += tid * 2;
    spawn_n[0] = n - 1; /* spawn */
    spawn_n[1] = n - 2; /* spawn */
}   /* sync */
```

Listing 1 – Recursive Fibonacci kernel

3.6.2 Argument specification

Arguments need to be specified in the CPU code so OpenCLunk knows how to invoke them. This is done in the constructor to simplify the resulting code. There are two types of arguments: parameters and plain arguments. Parameters get passed to all tasks of all task groups in the same way. These will be either constants or buffers. Readable buffers are just large parameters. Writable buffers are useful in cases that the tasks each write to disjoint regions, as is the case with matrix multiplication. Read-writable buffers are less often useful because of the synchronization model; two applications are in-place sorting and algorithms that allow opportunistic pruning (e.g. increasing a lower bound for a solution). Arguments are created similarly, where the value represents only the initial value sent to the root task. The return type is a template parameter, and it is required only to be a non-`void` type. The Fibonacci argument specification is shown in Listing 2. The class inherits from the `Alg_spec<Result>` class, indicating a return value type `unsigned`. This is the class that contains the various `add_arg()` and `add_param()` functions. `Fib`'s constructor defines a single `unsigned` argument with initial value `n`.

```
class Fib : public Alg_spec<cl_uint> {
public:
    Fib(unsigned n)
    {
        add_arg(static_cast<cl_uint>(n));
    }
    /* ... */
};
```

Listing 2 – Recursive Fibonacci argument specification

3.6.3 Reduction

The reduction operation is performed on the CPU, and so it is also a part of the `Arg_spec` subclass. OpenCLunk will first copy the results to a buffer, since they may have been rearranged. The Fibonacci reduction is shown in Listing 3, overloading a pure virtual function in the super class. The reduction is only the function `reduce()`, and its argument types match the template argument. Also shown on lines 11–18 are two other necessary functions (pure virtual in the super class) that describe the name of kernel function to be executed and the exact number of subtasks the kernel will spawn (if any).

```
class Fib : public Alg_spec<cl_uint> {
    /* ... */
protected:
    void reduce(
        const cl_uint *child_results,
        size_t len, cl_uint *result)
    {
        *result = child_results[0] +
            child_results[1];
    }
    std::string name() const
    {
        return "fib";
    }
    unsigned sub_tasks() const
    {
        return 2;
    }
};
```

Listing 3 – Recursive Fibonacci reduction, kernel name, and subtask count

The user code must then invoke the function, which requires only minimal work. Listing 4 shows the process of invoking OpenCLunk with the above Fibonacci example. Lines 1–2 initialize OpenCLunk and compile the list of source files. Line 4 creates the Fibonacci instance and line 5 begins the execution.

```
  const char *paths[] = { 'fib.cl' };
  Openclunk openclunk(paths, paths + 1);

  Fib fib(17);
5 unsigned result = fib.run(openclunk);
```

Listing 4 – Recursive Fibonacci invocation

4 Results

4.1 Evaluation Methodology

Ultimately, the best metric of OpenCLunk's performance is its speedup compared to sequential code. To this end, we implemented a number of benchmarks on the OpenCLunk framework and compared their performance with an equivalent version in C (similar to the C elisions for Cilk code). We expected to see low performance, and so we also chose to evaluate on benchmarks which expose the inefficiencies of the platform.

Our test platform consisted of a comparable CPU and GPU. Both were purchased at the same time, July 2009. The CPU was the AMD Phenom II 955. It was manufactured with a 45 nm process, had 4 3.2 GHz cores, and caches L1: 4×(64 kB + 64 kB), L2: 4×512 kB, L3: 6 MB. The GPUs were two ATI Radeon HD 4870 cards in CrossFire (equivalent to NVIDIA's SLI). The GPUs were manufactured with a 55 nm process, had 800 stream processing units 512 MB GDDR5 memory, and a PCI Express 2.0 ×16 interface.

4.2 Benchmarks

We ported or implemented a number of benchmarks to evaluate OpenCLunk. It was possible to port `fib`, `heat`, and `knapsack` directly from the examples distributed with Cilk-5.4.6. The `fib` benchmark calculates the Fibonacci sequence using the recursive algorithm. It is a useful benchmark because it does so little work per invocation that it is ideal for measuring overheads associated with the system. `heat` models heat diffusion. `knapsack` solves the 0-1 knapsack problem, where a thief is trying to fill her knapsack with the most valuable combination of items that does not exceed a specified weight. These three examples could be easily ported, with the main calculation going into the kernel and the reduction operation following simply from the return value of the function.

One interesting thing about `knapsack` is that it employs pruning: it keeps track of the best combination of items found thus far, and does not spawn new tasks with a given item combination if there is no way for it to be better than the previously found solution. In the original Cilk algorithm, this was done by way of a global where the write is a benign race condition. We employed the same method, but rather than using a global, we used instead a single-element read-write buffer. As it is only a hint to the algorithm to stop checking some combinations early, it does not matter in terms of correctness which thread succeeds in writing to the global if more than one tries in an iteration.

Several of our other benchmarks have Cilk analogs, but we re-implemented them rather than porting them. This was mainly due to some limitations in OpenCLunk as compared with Cilk, as previously discussed. The `n-queens` benchmark solves the problem where n queens must be placed on an n by n board such that none of them are in the same row, column, or diagonal. It is similar to the one distributed with Cilk, but instead of returning the first solution found, it finds all solutions. The `quicksort` benchmark performs an in-place quicksort, and is similar to cilksort. The `matmul` benchmark multiplies two matrices together, similar to Cilk's matmul, but with a slightly different algorithm.

Finally, we implemented one benchmark of our own. The `bucketfill` benchmark takes a matrix of integers, an old value, and a new value. If a starting point has the old value, the algorithm recursively changes that element and all neighboring elements with the same value to the new value. This is similar to the bucket fill tool in a paint program.

Benchmark	CPU Time	GPU Time	Speedup
`bucketfill(16, 18)`	3.85×10^{-6}	4.08	9.42×10^{-7}
`fib(10)`	4.80×10^{-7}	1.21×10^{-6}	2.30×10^{-6}
`fib(19)`	1.97×10^{-5}	9.62×10^{-1}	2.05×10^{-5}
`heat(medium)`	1.24	5.62	2.21×10^{-1}
`knapsack(11)`	3.53×10^{-7}	1.63	3.53×10^{-5}
`matmul(64)`	1.70×10^{-6}	2.39×10^{-1}	7.11×10^{-6}
`matmul(256)`	1.70×10^{-6}	7.49×10^{-1}	2.27×10^{-6}
`n-queens(4)`	2.23×10^{-5}	4.83×10^{-1}	1.82×10^{-5}
`n-queens(6)`	2.23×10^{-5}	4.83×10^{-1}	4.62×10^{-5}
`quicksort(1000)`	5.48×10^{-5}	1.10	5.00×10^{-5}
`quicksort(9000)`	5.60×10^{-4}	1.46	4.09×10^{-4}

Table 1 – Performance of OpenCLunk on the GPU as compared to a sequential version on the CPU. Times are measured in seconds.

4.3 Performance

We compared speedup to the sequential C elision running on the CPU. The times and speedups are shown in Table 1. Except for `heat`, `matmul`, and `quicksort`, all benchmarks are shown with the largest input size before they would need to start breaking apart thread groups; `matmul` fails sometime between 256 and 512, while `quicksort` fails sometime between 9000 and 10000. The speedups were all quite low, but there are a few interesting features to notice.

The benchmark that performed the best was `heat`, which we attribute to two qualities that it alone has. The only other floating-point benchmark was `matmul`, but the difference is that `heat` has better locality. The way they are written, each thread in the final thread group of `matmul` multiplies a subsequence of columns of one matrix by the entire other matrix. Each thread of `heat` in the final thread group gets their own column of the matrix, and does reads only on the two neighboring columns

Three of the algorithms performed better as the problem size increased. `fib`, `n-queens`, and `quicksort` all had higher speedups with larger input sizes. This was expected since larger task groups have higher throughput. On the other hand, `matmul` went slower with a larger input size. The difference between the two sets of algorithms is that the `matmul` does work only with the last task group, while the rest do work in all task groups. `fib` is the exception which does the absolute minimum amount of work in either case, but that also means it is balanced.

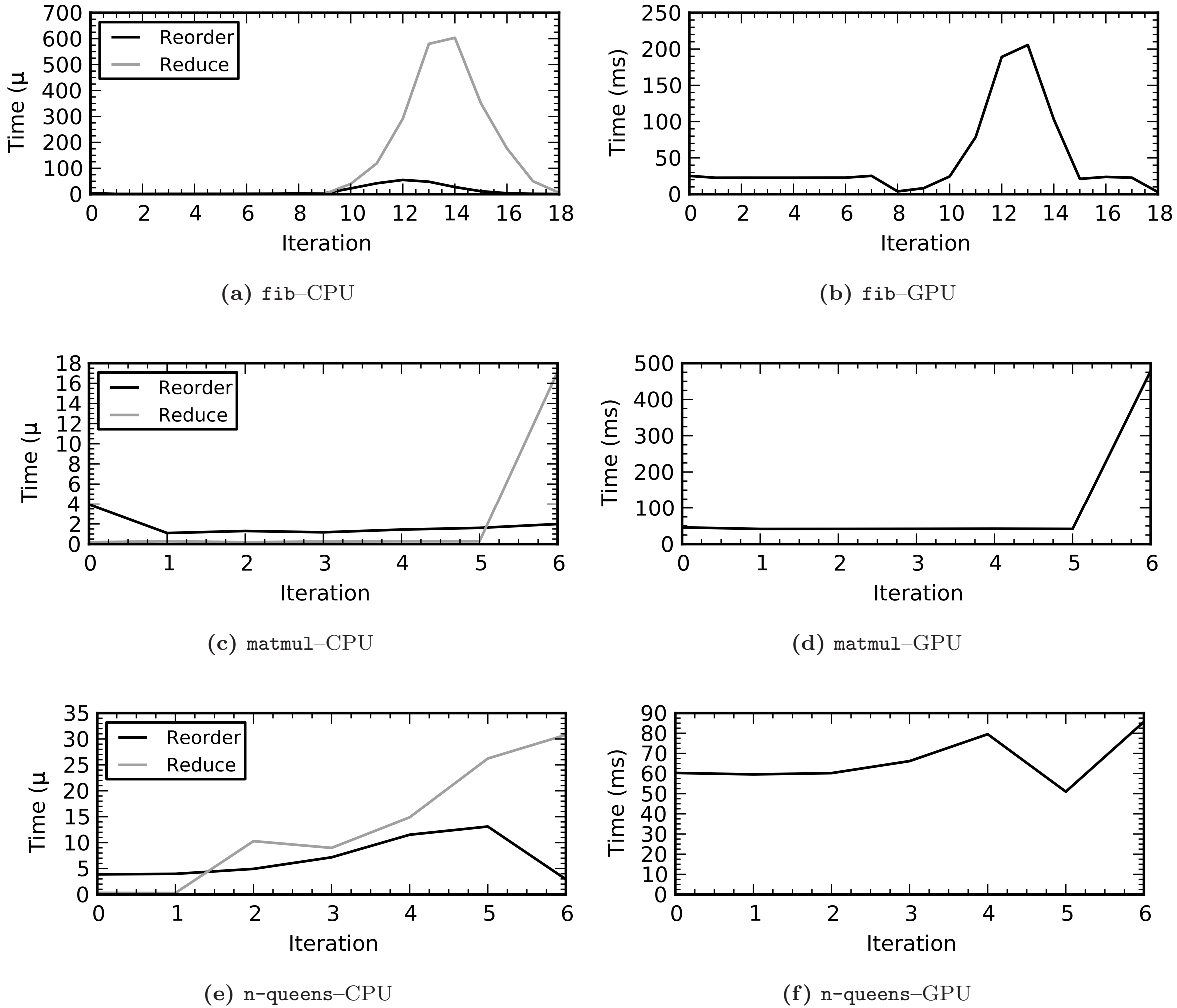
4.4 Overheads

To determine where the overheads are coming from, OpenCLunk was instrumented to give timing information. We looked at how the various phases of execution changed over time from multiple perspectives. There are three primary components to the execution of some algorithm within OpenCLunk's main loop: buffer reordering (CPU), reductions (CPU), and the kernel (GPU, including CPU time to add the job to the OpenCL command queue). We looked closely at only three algorithms. We chose `fib` since it is a minimal task-parallel function, `matmul` because it does not do any work until the final iteration, and `n-queens` because it is has many distinguishing features (work is done in the spawn and not the base case, it has many spawns per task, and tasks often exit early). These overheads are shown in Figure 1.

`fib` has a hump at the 13th and 14th iterations. All threads are still active, but afterward, many threads begin completing without spawning. The dip before the hump in Figure 1b was consistent; it occurs before the first tasks begin to complete without spawning (iteration 9). We attribute it to an artifact of OpenCL.

The behavior of `matmul` is expected: all tasks spawn except in the last iteration where none do. No reorders ever occur, reductions only happen at the end, and the GPU does not perform any work except at the last iteration.

The `n-queens` behavior is more varied. Its volatility comes from the tasks that die because they are infeasible. Reordering is a bigger issue with `n-queens`,



(a) `fib`–CPU **(b)** `fib`–GPU

(c) `matmul`–CPU **(d)** `matmul`–GPU

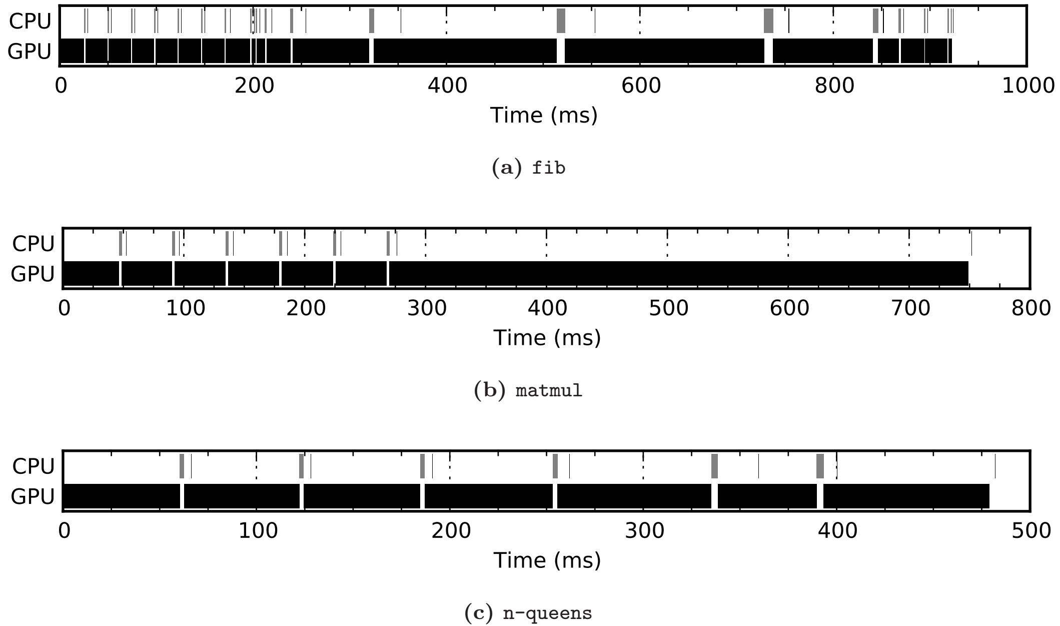
(e) `n-queens`–CPU **(f)** `n-queens`–GPU

Figure 1 – Buffer reordering and reduction overheads (CPU) and kernel execution time (GPU) throughout execution.

which is 4 µs at minimum (excluding first and last iterations) while `fib` is as low as 1 µs. This comes up because of the infeasible solutions that stop early. The increasing reduction costs is attributed to the increase in infeasible solutions as the number of tasks increases. Finally, the kernel execution time remains somewhat consistent. The variation on iteration 5 we attribute to subtleties of the n-queens problem.

The most useful view of the execution is in the timelines in Figure 2. Clearly, the CPU overheads are inconsequential, and the reductions could stand to be far less efficient without affecting anything. In the `fib` timeline the execution time of the kernels increases over time. Given that the body of the kernel is the minimum for OpenCLunk (takes one argument, usually spawns twice), the increased kernel execution time must be because of the data transfers. Further, the execution time can be seen in the first iterations, consistently near an average of 23.1 ms. Inexplicably, there are two intervals with periods 3.84 and 8.27 ms, corresponding with the dip in Figure 1b.

Even though it performed poorly, `matmul` had the expected behavior. There were a few short iterations where the work was merely divided, and then a long execution without having to cross back and forth between the CPU and GPU. Unfortunately, the execution time was just far too long, perhaps because there were only 64 threads executing that last iteration. Furthermore, about 40% of the overhead is attributed to overheads in crossing between the GPU



(a) `fib`

(b) `matmul`

(c) `n-queens`

Figure 2 – Timelines of the execution separated by CPU vs. GPU. In the CPU timeline, reordering and reduction steps occur close together, reordering usually the thicker band.

and CPU, so there many factors preventing speedup.

Execution time did not vary for `n-queens` despite the large number of spawns at each stage. It was expected that the kernel execution times would grow much faster than `fib`, so the difference may be due to abandoning infeasible solutions. Aside from the large spawn count, the overheads in the algorithm come from the array argument. It is required to be copied in the kernel for each spawn, and it consumes a lot of memory. This is why so few iterations were possible, but it also explains the large minimum task execution time.

5 Discussion

As seen in Section 4, OpenCLunk failed to perform well. We will discuss several reasons for this, followed by a brief discussion of possible improvements that could lead to better performance in OpenCLunk.

5.1 Performance of OpenCLunk

There are several reasons why OpenCLunk failed to attain good speedup. The main problem is that there is a fundamental mismatch between the strengths of Cilk and the strengths of the GPU. Cilk performs well on benchmarks with a large number of `spawn`s, because in the common case, they are not much more expensive than ordinary function calls. Therefore, programs written for Cilk tend to `spawn` many more times than there are threads, because it enables load balancing. Each task tends to do only a small amount of work. Although OpenCLunk also needs many `spawn`s in order to take advantage of the large number of threads available on the GPU, they are more expensive than in Cilk, as shown by both the amount of overhead and the mandatory synchronization inherent in generations of OpenClunk `spawn`s. Hence, to get good performance, there must be more computation in each `spawn`ed function to amortize the higher cost. The benchmarks that we tested on have little computation per kernel, and are therefore not well suited for OpenCLunk. In general, tasks will not have much computation; if there is a lot of computation per kernel, the problem could easily be adapted to use a data-parallel model and OpenCLunk would not be necessary.

Another reason that we did not attain high speedups was because the benchmarks we tested on did not, in general, exploit the strengths of the GPU. In addition to the small amount of computation per ker-

nel, most of the benchmarks use integer rather than single-precision arithmetic, and we made no use of the fast transcendental functions built into the GPU. For both of these cases, it could be argued that we should simply test OpenCLunk on benchmarks that would be expected to perform well, such as ones with large amounts of computation in each step and which use transcendental functions. However, those types of problems can already be implemented efficiently and easily on the GPU. It is much more valuable to examine the performance of benchmarks that could not previously be easily run on the GPU.

5.2 Potential improvements

There is room for OpenCLunk to improve, but we expect that the improvements would not be enough to make OpenCLunk perform well. Eliminating the cost of switching between the CPU and GPU would be key. One improvement would be to keep the spawn arguments on the GPU; it would make debugging harder, but there is definite room for improvement for algorithms with large arguments.

Reducing the number of times tasks actually execute on the GPU would reduce the overall cost of CPU–GPU switches. One way of doing this would be to 'unroll' the recursion one or two steps; instead of spawning just two tasks, `fib` might spawn four or eight.

Another method would be to more accurately find the resource limit. Currently, the resource limit is reached because the spawn argument buffers get too big. Some algorithms might be rewritten to switch to a different kernel on the last iteration that can fit within the resource limit. This final kernel would compute the result without further spawning (perhaps using a different algorithm than it otherwise would), so large spawn argument buffers would be unnecessary. Since no further spawns would occur, there would be no further CPU–GPU crossings, allowing the costs of OpenCLunk to be amortized. It could also achieve a better thread-to-memory ratio because the memory would no longer be the limiting factor, and so it would better utilize the capabilities of the GPGPU.

6 Conclusion

We showed that it is possible to write a Cilk-like task-based parallel programming framework for the GPGPU. We were able to port Cilk benchmarks and run them on the GPU, obtaining correct results. In the case of `heat`, the performance was actually nearing baseline, and with some of the above improvements, it could feasibly see speedup. However, in all other cases, we found that the performance was unusably bad, indicating that it would not be worthwhile to continue with the project or to write a compiler for OpenCLunk. The reasons that the framework failed can mostly be attributed to the fundamental incompatibility between the advantages of the GPU and the advantages of Cilk. Furthermore, the high transfer time between the GPU and CPU created an insurmountable amount of overhead.

It would take large architectural changes to graphics processors to make Cilk-like parallelism on GPUs a reality. Higher transfer speeds would reduce the time to execute task groups by a constant factor. Lower delays would also improve performance by reducing the minimum execution time of task groups. If GPGPUs had the ability to create tasks themselves, that would fix the issue as well; that would be nothing more than a shift towards the CPU model. Without these changes, the Cilk-like model will remain infeasible on GPGPUs.

References

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

[2] Khronos Group. OpenCL – The open standard for parallel programming of heterogeneous systems. `http://khronos.org/opencl/`.

[3] Khronos Group. The OpenCL specification 1.0. `http://khronos.org/opencl/`.

[4] Lee, S., Min, S.-J., and Eigenmann, R. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 101–110.

A General OpenCLunk example

What follows is a general example that uses more of the OpenCLunk interface. The interface was developed while implementing the various benchmarks in our framework, so the following illustrates most of the supported ways for arguments to be used.

Parameters are values or buffers that are the same across all tasks. They can be values, readable buffers, and writable buffers. Listing 5 shows the arguments for matrix multiplication, and it is a mostly comprehensive example. Lines 7–10 specify read-only buffers, the operands to the multiplication. `add_param()` can be used similarly for a buffer argument; these require more careful indexing in the kernel function. Lines 8–9 show a writable buffer, the result of the multiplication. Lines 13–15 are the dimensions, and lines 16–17 specify two column indices, used in the work division.

In the kernel (in a separate file), arguments are all in a precise order, partially determined by the `Matmul` constructor. First are the six parameters in the same order they were defined in the constructor; there are zero or more in general. Then comes an `unsigned max_tid`, the global work size, and it is always present. The two arguments come next in the same order they were defined in the constructor; in the general case there are zero or more. `result` comes next (it will not be used, but it must have non-zero size), followed by `spawn`; they are always present. Last are the spawn arguments corresponding with the arguments on lines 30–31.

Lines 41–48 show how `Matmul` would be called. In many cases, including this one, using code that invokes OpenCLunk can be almost completely transparent.

```
class Matmul : public Alg_spec<cl_uchar> {
public:
    Matmul(const cl_float *A,
        const cl_float *B, cl_float *C,
        cl_uint m, cl_uint n, cl_uint o)
    {
        add_param(A, m*n*sizeof(cl_float),
            Argument_buffer::read);
        add_param(B, n*o*sizeof(cl_float),
            Argument_buffer::read);
        add_param(C, m*o*sizeof(cl_float),
            Argument_buffer::write);
        add_param(m);
        add_param(n);
        add_param(o);
        add_arg(static_cast<cl_uint>(0));
        add_arg(o);
    }
    /* ... */
};

void __kernel matmul(
    __constant float *A, /* read-only */
    __constant float *B, /* read-only */
    __global float *C,   /* write-only */
    unsigned m,
    unsigned n,
    unsigned o,
    unsigned max_tid,
    __constant unsigned *arg_c0,
    __constant unsigned *arg_c1,
    __global unsigned char *result,
    __global unsigned *spawn,
    __global float *spawn_c0,
    __global float *spawn_c1
)
{
    /* ... */
}

void matmul(Openclunk &openclunk,
    const cl_float *A, const cl_float *B,
    cl_float *C,
    unsigned m, unsigned n, unsigned o)
{
    Matmul matmul(A, B, C, m, n, o);
    matmul.run(openclunk);
}
```

Listing 5 – Matrix multiplication arguments