CS133: DNArm - Final Report

by Stephen Chen, Joey Degges, Jeffrey Finch, Kalvin Hom, Noah Kagan and Joel Miller

Guide to Source Code

Base packing:
This portion is completed and compiles successfully. Source file: DNArm/src/ctb.c (Char To Binary is the acronym as it converts ASCII symbols to a packed binary representation). This uses a thread pool abstraction provided by a threading library called loomlib (http://www.github.com/jdegges/loomlib) which was previously written by Joey. loomlib provides a pthread_create like interface that adds new functions and data to an asynchronous queue which is serviced by a variable amount of threads that are created only once. The thread functions in ctb.c are convert (line 44) and convert_inv (line 95). convert takes an ASCII encoded buffer and strips each character down into two bits and convert_inv does the inverse operation: takes in a binary coded buffer and fills in the proper ASCII bits. Simple locking is done on the output file stream to ensure that all threads output in the correct order.

Database:
The database code compiles successfully. The database functions properly, for both reading and writing. Writing is done only sequentially, and reading is done in parallel using pthreads. DNArm/src/cldb.c:cldb_get invokes DNArm/src/cldb.c:cldb_wait on cache misses, which is the function responsible for maintaining barriers to have threads wait and grab information from the cache blocks at the appropriate time. The precache function also uses a thread that waits for a signal to grab the next cache block.

Index generation:
The index generation code compiles successfully. It takes as input the base packed genome file and parses the entire file, determining all unique 16 base length keys and the corresponding locations of those keys in the genome. The way we decided to parallelize this section was to read sections from the file and distribute the work corresponding to each section to a different thread. This was accomplished using POSIX threads.

Coarse-grain matching:
The coarse-grain matching code compiles successfully. It takes as input a number of reads and a structure containing database information and after computation, passes the results to the fine-grain matcher. After experimenting with different levels of parallelization and comparing results, we decided that applying the parallelization at the top-level was the best option. As it uses OpenMP, the parallel pragma is located around the outer for loop that breaks up the reads into smaller chunks. Each thread receives a chunk of reads, determines the likely locations of matches, and sends the fine-grain matcher a pointer to an array of structs containing the results of each read.

Fine-grain matching:
The iterative fine-grain matching code compiles successfully, and the gpu-parallelized code will be completed in the next couple days. The fine-grain matcher (FGM) is called by the coarse-grain matcher, when enough read sections have had possible location lists built to warrant farming the workload out to the GPU cores. There is some initialization that occurs to set up the reference genome in the GPU memory and the kernels, but this only occurs once - copying the reference genome on every function call would be a formidable performance hit. When enough work-items have been accumulated, the fine-grain matcher kick-off function is called, which proceeds to set up the data on-GPU and launch the threads.

Fine grain matching, as it sounds, entails taking a small locality of the reference genome and producing an exact match to a read sequence, noting where any mutations, insertions, and deletions occur. In the code, this is achieved by investing a list of possible location matches that are passed in by the coarse grain matcher.

The actual process involved for finding a match and calculating the mutations is rather simple. Since we are dealing with a compressed data set, with each nucleotide represented by two bits, a 32-bit unsigned integer can, of course, represent a sequence of 16 bases. Though there are some complications and special cases that arise from "unaligned" data - that is, sequences that do not start on a boundary of an unsigned integer storage bin - this approach is overall rather efficient, especially when considering the presence of insertions and deletions.

When given a possible matching location, the FGM creates a set of skewed reference sequences that have been shifted +/- 3, 2, 1, or 0 places, and creates a corresponding "diff sequence" (of the form 0_0_..., where '_' is either 1 or 0 given a mismatch or a match, respectively). In the most trivial case (without any insertions or deletions or insertions), we can simply count the number of differences and check against our threshold value. In the case of an insertion or deletion, the two sequences with the lowest number of differences are investigated. If the start of the higher index sequence or the end of the lower index sequence matches very closely, then an insertion has been detected; if the start of the lower index sequence or the end of the higher index sequence matches closely, a deletion has been detected. By using this information, the exact "change-point" - that is, the location at which the lead sequence will stop matching closely, and the follow sequence will start matching - can be found. From this point, finding the actual mutations, insertions, and deletions is a fairly trivial exercise. A deeper understanding of the algorithm by inspecting the code found in fgm.c (the sequential version), which has been commented heavily for explanation.

OpenCL Research:
There are five components to the OpenCL code, each utilizing different techniques. These techniques are spread out among three C source files, and several OpenCL source files, and all come packaged as part of the cgmtest source folder with its own separate testing harness. Each OpenCL implementation attempts to optimize the course grain matching functionality of our project, but breaks it down into comparing two lists of randomized indexes to simulate the database lookup. The techniques are: simple "linear" search, binary search, constant memory caching, image memory caching, local memory caching. In addition, each of these techniques were configured to run on both the CPU and the GPU. The source code for all of these techniques compiles correctly, given that the proper packages are set up in your environment. To test the OpenCL code, you need at a minimum either the ATI Stream v2.1 OpenCL library, or a recent NVIDIA graphics card with recent drivers installed. For a full list of compatible graphics cards, see Appendix A of:

http://www.nvidia.com/content/cudazone/download/OpenCL/
NVIDIA_OpenCL_ProgrammingGuide.pdf

Note that not all cards, even if listed on that document, will work completely with all of the OpenCL code, as only some of them have image memory support. If testing on lnxsrv, it is required to use ATI Stream v2.1 and to run on the CPU device (see readme in the package for details).

Although all of them compile, they don't produce consistent results. Due to the nature of OpenCL being device- and implementation-dependent, much of the OpenCL programs have inconsistent behavior. In general (such as on lnxsrv), image memory caching and running on the GPU are not supported. If the system has a recent graphics card installed with the proper drivers, then running the OpenCL code on the GPU device works properly. Unfortunately, even some of the latest cards don't have image memory support (some Radeon 5870 HD cards lack the proper drivers, strangely). When they do compile and run, however, all of the OpenCL implementations besides image memory caching work as intended and produce correct results. Image memory optimizations had to be sacrificed due to problems testing the kernels, as are described later. All of the kernels are located in external source files with the .cl extension.

Project Breakdown

Base packing:

The packing operation at a character level is quite trivial so any parallel attempt had to operate on somewhat large batches of input. The most intuitive method for achieving this seemed to be to dedicate one thread to reading in input and have one thread crunch away on each block of input. This way the input can be read as fast as possible helping to minimize the IO bottleneck while useful work is being done. Locks were needed around the output file stream so that each thread writes its output in the correct order and doesn't trample on any other currently writing thread. A global output ID counter is maintained that counts from zero and the dispatcher thread, the one that reads inputs and pushes work items into the pool, tags each input block with a sequential number. When a worker thread finishes processing its input block it does a conditional wait until the global output ID counter. When the global counter equals the threads output ID it writes to the output stream and then increments the global ID counter (appropriately locking with a mutex of course). This section was worked on by Joey Degges and was completed to specification.

Database:

Originally, the database was implemented using the GNU dbm. However, after testing, it was realized that the database size was limited to 4GB, far less than what we needed for the genome. We then moved on to trying the Tokyo Cabinet, but that was was too inefficient in performance. The final solution was to create our own database implementation.

In our database implementation, a scheme similar to the way caches work was created to efficiently read data sequentially from blocks. The db works under the assumption that reads will always be done sequentially, allowing greater efficiency to be allowed in the implementation. The db contains an L1 block, which has 256 pointers to different L2 indexes. The L2 index blocks each contains 2^24 items, each pointing to a section of the L2 data where the result the user wants is. Altogether, the L1 block is 4KB, each L2 index is 128MB, and the L2 data size varies on the number of values match with the key.

To facilitate efficient reading from the database, a L2 index and data block is loaded on the RAM. When the user makes a request and the corresponding L1 pointer matches the current loaded block, reading the data is efficient and can be done concurrently. As this happens, a single thread also pre-caches the next L2 index and data block into RAM.

On a cache miss, where the L1 pointer references a block past the current loaded one, the thread that made the request hits a barrier and waits. When all of the threads hit this barrier, it signifies that no thread needs the current cached block anymore. At this point, the pre-cached block gets switched with the current loaded block, and the threads can proceed reading again. A signal is also sent to the pre-caching thread to grab the next block to pre-cache. The database portion was done by Joey and Kalvin.

Index generation:

The index generation was originally implemented sequentially, then converted to an implementation using pthreads. We read from the binary file 1024 32-bit integers at a time. Iterating through this array, we concatenate two 32-bit integers at a time to form a 64-bit integer (string of bases). From this 64-bit integer we determine all sequences of 32 bits (16 bases), which are our possible key values, as well as the corresponding position of each key. The position is determined by an offset calculated from the position in the binary file added to the offset of the position in the array. Once we have determined a (key, value) pair, we insert this pair into the database where either a new key is inserted, or a new value is added to the corresponding keys value list

Coarse-grain matching:
Originally we planned to have this part run using OpenCL on the GPU. However, after experimentation on lnxsrv as well as one of our own machines, we realized that for the amount of data we expected this part to process, it was not worth the overhead of copying data to the GPU. We also attempted a couple of different ways of data decomposition which proved unsuccessful as only polynomial runtime was possible compared with the linear time of the sequential algorithm. At this point, we decided we were better off using task decomposition and running the coarse-grain matcher on as many reads as possible in parallel. No synchronization is needed at this level because no data is being written to shared memory. Given more time, it may have been more efficient to complete this portion using pthreads, but the OpenMP version still can process large amounts of data in reasonable time. This portion of the project was completed by Jeff Finch.

Fine-grain matching:
Like the coarse-grain matching algorithm, this part was originally aimed at running on the GPU as well. However, we ran into some major issues with pointer address spaces. The algorithm used to precisely detect mutation relies on being able to manipulate pointers in rather fine detail; in order to parallelize the dataset efficiently, we planned on batch-processing 1024 possible fine-grain matches at once. However, OpenCL does not allow the assignment of private address space pointers to global memory regions. This restriction would have necessitated a substantial rewrite of the code. Additionally, we already were wary of implementing the fine-grain matcher as a kernel in OpenCL due to the rather high degree of complexity and large amount of branching. When these two complications came together, we decided that it would be more efficient (in terms of both our time and of gpu time) to pursue a different method of parallelization.

This isn't to say that parallelizing this application on a GPU is not possible; indeed, if the fine grain matcher didn't deal with insertions and deletions, it would be a relatively minor issue to correct, and it would likely be very efficient. However, the detection of insertions and deletions necessitates a large amount of branching, and thus isn't the most efficient algorithm to parallelize on a gpu, since the cores must operate in lockstep.

OpenCL Research:
In essence, we decided to parallelize the code in OpenCL as an experiment to see if it would be faster than OpenMP for our purposes. OpenCL uses a queue to maintain a list of tasks to perform, such as copying data to buffers, setting arguments, and more. For all of the host programs (C programs) that organized the OpenCL kernels, many barriers were needed to ensure that the queue processed all the tasks currently enqueued before continuing on with other tasks. In addition, in the local caching implementation, barriers were needed to guarantee that the cache was prepared before the various work-groups in the kernel began using it.

There was a lot of ground to cover with OpenCL. After we determined that the original implementation of using global memory buffers required too much overhead to prove useful, we tried to find other ways of optimizing the kernels in hopes that it would speed up the execution. There are several types of memory available to the kernel during execution.There are several types of memory in OpenCL. Originally we used global memory which is stored on the host device (CPU). This is the most convenient memory to use for communicating with the host process as both the host and kernel can acceess it directly. When run on the cpu, this is equivalent to accessing any other memory

location in ram. However, when the kernel is run on the gpu device, data access becomes extremely slow by several orders of magnitude compared to other methods. These other methods are any technique that takes advantage of memory located locally on the gpu. First we tested with using image memory. There are pros and cons to image memory. On the plus side, reads to the image automatically trigger caching on the gpu device. However, the drawbackks are numerous. First, the host must use an abstracted form to store data in a pseudo image format. That is, the host can use any data type, but must trick the kernel into thinking the image is of some specific format. For our purposes, we used integers to fill in data for RGBA values of the image's pixels. To access the image in the kernel, one must use abstracted methods such as read_imageui() which returns an int4, OpenCL's built in integer vector datatype which stores 4 unsigned integers. This function is slower than direct memory access, and, unfortunately, we were unable to do enough tests with it to make any definitive statements about our implementation for reasons stated above.
The next memory method we used was similar to the original global simple "linear" parallel method. However, I found out that by declaring the incoming data argument as a constant allowed the data to be cached in the same local buffer as reads to images. This turned out to be implementation specific, and with ATI Stream on lnxsrv we noticed no benefits, or inconsequential ones at the least. This made it difficult to determine if the caching was coming into effect, especially since there is no debugging during the kernel's execution.

Lastly, in an effort to force the gpu device to cache our dna reads in the most efficient and specific way possible, I made another kernel which stores memory in the local memory space given to groups of computational units. This is analogous to an L2 cache on modern processors. Although the division of the data is left up to the gpu and library, we determined the amount of space needed to cache during runtime. OpenCL divides the data space to distribute among groups of computational units, which is reffered to as a work-group. These work-groups, composed of work-items, all share a single memory cache. to take advantage of this, I had one work-item read in the data from the constant argument passed in by the host and cache only the segment of data relevent to its work-group in thay group's local memory. This was done in a critical section, followed by a barrier. Then, each work-item performed its usual task of comparing items, this time using the cache. Unfortunately, in the end, even running on a gtx 285 with NVIDIA OpenCL drivers installed, the sequential implementation was most efficient. It seems that the more elegant and complex the OpenCL solution, the slower it got. Each work item is doing relatively little work, usually just a two integer comparison, yet it takes a massive amount of set-up to reach this task, so the parallelization doesn't pay off. However, as a nice proof of concept, the local caching had great improvements on the GPU compared to its CPU counterpart.

This part was implemented by Noah Kagan.

Bugs and Challenges

Base packing:
The largest challenge for this part of the project was figuring out how the bits are laid out in memory. Initially the bit shifting in the packing and unpacking stages was quite inconsistent but after writing some simple straightforward tests the bit orderings was figured out. Other than that it was quite straightforward and most of the parallel programming techniques from the pthread section of class could be directly applied.

Database:
In order for our project to work efficiently, it is necessary to have a fast, parallelizable database. The initial challenge was to find a suitable database for our needs. We attempted to use the GNU Database Manager, the Tokyo Cabinet, and others, but all had their limitations that made them unsuitable for our needs. For example, the GNU dbm limited the DB to 4GBs, which was not enough to store our genome, and the Tokyo

Cabinet was optimized for a general case that made it slower for our specific needs. In the end, we had to implement our own database to meet our requirements.

Index generation:
For the most part, the key generation work was fairly simple but Joey did catch a small bug in the code. Because we used 2 bits to represent each base, the algorithm was supposed to increment by 2 bits per iteration, however the initial code constructed keys using a sliding window algorithm that shifted only 1 bit per iteration.
Due to troubles with the database, we have currently been running sequential index generation. This is because we currently do not support parallel writes to the database.

Coarse-grain matching:
For the most part, there were not bugs within the coarse grain matcher. Most of the problems arose because the sequential algorithm was performing better than the parallel algorithms. Originally, when this part was attempted in OpenCL it was difficult getting accustomed to GPU programming. A lot of function calls were necessary before and after executing the kernel in order to set up the required structures and dismantle them afterwards. In addition, it was harder to debug the OpenCL kernel than C code as it was hard to tell exactly where the problems were occurring.

After we decided to switch to task decomposition in OpenMP the parallelization became much simpler using just a dynamic parallel for loop. The OpenMP debugging was simpler than OpenCL but still more complex than sequential code as it was hard to determine which thread was causing a segmentation fault. Eventually, the problem was pinpointed to a logic error and was simply corrected. The main thing this part of the project shows is that parallelization does not always mean improved performance. The parallelization needs to be done in an optimal way and it is very important to consider any added overhead.

Fine-grain matching:
For the creation of the basic sequential algorithm, there was a rather large number of bugs that had to be solved. this is mainly due to alignment issues - read sequences not lining up with data bin boundaries - and shifting the diff sequence properly when detecting insertions and deletions. These bugs were particularly troublesome because fixing one often created another.

During the implementation of the multi-threaded algorithm in OpenCL, as mentioned before, we ran headlong into the issue of global/private address space conflicts, which essentially invalidated a lot of the shortcuts and pointer tricks we used in our algorithm. However, we also came up against some SDK peculiarities that made finding bugs an incredible pain. When trying to do the runtime compile of our kernel on our Radeon HD4850-based platform (using the ATI OpenCL SDK), we had compilation errors, but any attempt to retrieve the build logs from the runtime compiler generated an additional error - basically, we couldn't retrieve any useful information from the runtime-build failure. After many hours and much tweaking, we decided to try it on a different platform - a GeForce 9500GT-based platform, running the Nvidia OpenCL SDK. While this still gave a compilation error, it actually produced a build log that we could use and extract information from. This, of course, led to our realization that the address space conflicts were going to be a major stopping point; however, this episode also demonstrated how inconveniently buggy some cutting-edge SDKs, such as those designed to take advantage of GPU parallelism, can be.

At this point, we stuck with a batch-processing approach, but shifted to openMP to parallelize the algorithm. This approach was relatively straightforward, as we only had to make a simple batch-wrapper for our matcher, and include an "omp parallel for" directive.

OpenCL Research:
One of the largest obstacles was learning OpenCL individually. It is an open specification library, so learning the from the basic specification wasn't enough. I had to research the implementations of the library by both ATI and NVIDIA and how different devices have different behaviors, especially with regards to memory management. Also, testing the OpenCL implementations was a huge issue. Testing for correctness (besides image memory) could be done on lnxsrv at UCLA. However, that had to be done using kernels running on the CPU, which are not able to take advantage of the GPU's parallel processing power. We did have some access to graphics cards capable of running OpenCL, but not often enough, and it often had more issues running OpenCL than lnxsrv, even on the CPU. I originally thought my own graphics card was recent enough to be used, but it turns out that it didn't support OpenCL for some reason. In the end, this meant we had to scratch the image memory optimization because we didn't have enough time for testing, and has yet to be run to produce correct results.

Another large setback was a strange error we had with github. For some reason, the times on the commits from some of our members were recorded wrong. We even had people pushing commits in the future! At one point, this confused the repository so much, that a pull erased the local cgmtest folder (where the OpenCL research was being done). Luckily, an older version was visible in an older commit, but it wasn't accessible through git due to the commit timing issue, so it had to be hard copied from the source and re-pushed.

Final Results

Base packing:
The parallel results were very similar to the sequential results. After further inspection it was realized that the runtime of the packer was equivalent to the time it takes to copy the input source file from one disk to another. At this point there is obviously very little room for improvement as the code is entirely IO bound. When the input source file was cached in memory a modest improvement was seen since the IO bottleneck was alleviated to some extent.

Size of unpacked genome
```
$ du -sh /q/0/genome.unpacked
2.6G    /q/0/genome.unpacked
```

Run time with one thread:
```
$ echo 1 > /proc/sys/vm/drop_caches
$ /usr/bin/time -v ./src/ctb -i /q/0/genome.unpacked -o /q/1/genome.packed
-p 1
        User time (seconds): 20.08
        System time (seconds): 4.88
        Percent of CPU this job got: 53%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:46.35
$ du -sh /q/1/genome.packed
664M    /q/1/genome.packed
```

Run time with four threads (Quad Core machine):
```
$ echo 1 > /proc/sys/vm/drop_caches
$ /usr/bin/time -v ./src/ctb -i /q/0/genome.unpacked -o /q/1/genome.packed
-p 4
        User time (seconds): 28.14
        System time (seconds): 26.56
        Percent of CPU this job got: 121%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:45.10
```

```
$ du -sh /q/1/genome.packed
664M    /q/1/genome.packed
```

Time to copy unpacked genome:

```
$ echo 1 > /proc/sys/vm/drop_caches
$ /usr/bin/time -v cp /q/0/genome.unpacked /q/1/
        System time (seconds): 8.20
        Percent of CPU this job got: 12%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 1:03.74
```

Database and Index generation:
Building the database index requires two stages. First the geometry of the genome has to be computed. Here the geometry is the number of values associated with each key. This is necessary in order to optimize the real index generation by preallocating the exact amount of space for each key. The geometry database generation is carried out in DNArm/src/genome-geometry.c and uses the simple database defined in DNArm/src/cvdb.h. Ultimately a 16GB geometry file is created. It can hold 2**32 32 bit unsigned integer (unique key, value) pairs (2**32 * 32 bits = 16GB). This takes approximately 30 minutes to generate.

The second stage is the actual genome index generation. Here the geometry database is used along with the packed genome to build a larger database containing every (key, position) pair present in the genome. The source files for this generation are DNArm/src/genome-index.c. Most of the logic for both generating and querying this database are defined in DNArm/src/cldb.h. This operation takes approximately 1 hour and 20 minutes. Ultimately a 44GB index file is created. It can hold a maximum of 2**32 32 bit non unique unsigned integer keys. In our case only approximately 3 billion values will be inserted so 2**32 * 64 bits (pointers to each key's value list) + 3 billion * 32bits = 44GB.

Coarse-grain matching:
Original testing to decide between parallelization types (lists of 512 - max dimension on GPU) :

- Lnxsrv02 - 8 cores, 8 computation units
  - Sequential: 0.012700 ms
  - Parallel Binary Search: 0.281800 ms
  - Simple Parallel: 1.241550 ms
  - OpenMP: 0.035000 ms
- GTX 285 - 8 cores, 30 computation units
  - Sequential: 0.012700 ms
  - Parallel Binary Search: 0.129200 ms
  - Simple Parallel: 0.128550 ms
  - OpenMP: 0.027400 ms

Final testing on OpenMP implementation (50000 element lists):
Sequential:

```
bash-3.2$  /usr/bin/time -v ./cgm 3000000000 50000 1 25 100000

*****Result*****:
Average Time (w/o generating lists): 0.000000
Average Time (w/ generating lists and overhead): 3.573788
Total Time (w/ generating lists and overhead): 357378.848000
Command exited with non-zero status 255
        Command being timed: "./cgm 3000000000 50000 1 25 100000"
        User time (seconds): 339.96
        System time (seconds): 17.35
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 5:57.38
        Minor (reclaiming a frame) page faults: 16300303
        Voluntary context switches: 4
        Involuntary context switches: 550
        Page size (bytes): 4096
        Exit status: 255
```

Parallel:

```
bash-3.2$  /usr/bin/time -v ./cgm 3000000000 50000 8 25 100000

*****Result*****:
Average Time (w/ generating lists and overhead): 0.456442
Total Time (w/ generating lists and overhead): 45644.220000
Command exited with non-zero status 255
        Command being timed: "./cgm 3000000000 50000 8 25 100000"
        User time (seconds): 341.94
        System time (seconds): 20.75
        Percent of CPU this job got: 794%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:45.67
        Minor (reclaiming a frame) page faults: 8974896
        Voluntary context switches: 77693
        Involuntary context switches: 9124
        Page size (bytes): 4096
        Exit status: 255
```

Final testing on OpenMP implementation (2 element lists):
Sequential:

```
bash-3.2$ /usr/bin/time -v ./cgm 3000000000 2 1 25 1000000000

*****Result*****:
Average Time (w/ generating lists and overhead): 0.000331
Total Time (w/ generating lists and overhead): 331157.457000
Command exited with non-zero status 255
        Command being timed: "./cgm 3000000000 2 1 25 1000000000"
        User time (seconds): 331.10
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 5:31.16
        Minor (reclaiming a frame) page faults: 190
        Voluntary context switches: 5
        Involuntary context switches: 843
        Page size (bytes): 4096
        Exit status: 255
```

Parallel:

```
bash-3.2$ /usr/bin/time -v ./cgm 3000000000 2 8 25 1000000000

*****Result*****:
Average Time (w/ generating lists and overhead): 0.000068
Total Time (w/ generating lists and overhead): 68481.731000
Command exited with non-zero status 255
        Command being timed: "./cgm 3000000000 2 8 25 1000000000"
        User time (seconds): 547.32
        System time (seconds): 0.01
        Percent of CPU this job got: 799%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 1:08.48
        Minor (reclaiming a frame) page faults: 233
        Voluntary context switches: 19
        Involuntary context switches: 15685
        Page size (bytes): 4096
        Exit status: 255
```

Even on short lists, the final OpenMP implementation performs approximately 5 times better than the sequential version. On longer lists, it has close to 8 times better performance than sequential.

Fine-grain matching:
Unfortunately, we did not get a parallelized version of the fine-grained matcher working, or integrated with the rest of the project. This was specifically due to a high degree of focus on trying to get the OpenCL implementation to work, which consumed a rather large amount of time. We worked up to the wire trying to debug and tweak the OpenCL version to work, but to no avail; with a couple hours left, we decided to throw it into OpenMP just to have some parallelism, but unfortunately that implementation is buggy. Stopping the OpenCL effort a few hours earlier would definitely have been a great idea; as it stands, the sequential version works great, but the parallelized versions do not work. The OpenCL implementation will pre-compile, but will fail at runtime compilation of the kernel due to memory address space conflicts, as mentioned before.

Though it is only sequential, it is still quite fast - a 10000x loop of a 34 test case regimen can be completed in about 1.1 seconds. Since there's no data dependence between calls to the fine grain matcher, parallelizing calls to it would theoretically have a direct and linear effect on the execution time - 4 cores would probably have sped it up ~4x, 8 cores by 8x, etc.

OpenCL Research:
The lnxsrv testing I was able to do myself, but to run it on the GPU, Joey lent me the use of his brother's linux box with a gtx 285.

<u>Running on lnxsrv with ATI Stream v2.1:</u>

```
Running tests on CPU...
All tests run with key length of 16
Times are averaged over 10 randomized runs
Note that saturated lists are ones with many matches.

==============================================

*** Key Maximum: 32 List Size: 100 Saturated:

Sequential: 0.004300
Binary Search: 0.196200
Simple Parallel: 0.399700
Constant Cacheing: 0.439700
Local Cacheing: 2.936200

*** Key Maximum: 1000 List Size: 100 Unsaturated:

Sequential: 0.003800
Binary Search: 0.244700
```

```
Simple Parallel: 0.492300
Constant Cacheing: 0.420400
Local Cacheing: 3.051300

*** Key Maximum: 3276 List Size: 32768 Saturated:

Sequential: 0.002500
Binary Search: 0.197300
Simple Parallel: 0.704000
Constant Cacheing: 0.418200
Local Cacheing: 3.488200

*** Key Maximum: 1048576 List Size: 32768 Unsaturated:

Sequential: 0.006700
Binary Search: 0.207800
Simple Parallel: 0.399100
Constant Cacheing: 0.420300
Local Cacheing: 2.897800

==================================================

Running tests on GPU...
All tests run with key length of 16
Times are averaged over 10 randomized runs
Note that saturated lists are ones with many matches.

==================================================

*** Key Maximum: 32 List Size: 100 Saturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: GPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: GPU.


*** Key Maximum: 1000 List Size: 100 Unsaturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: GPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: GPU.

*** Key Maximum: 3276 List Size: 32768 Saturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: GPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: GPU.

*** Key Maximum: 1048576 List Size: 32768 Unsaturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: GPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: GPU.
```

<u>Running on a gtx 285 with NVIDIA OpenCL Drivers:</u>

```
Cleaning folder.
Building test programs...
```

```
gcc -o ./build/cgmtest -I/home/jdegges/local/include -L/home/jdegges/local/lib -I.
cgmtest.c -lOpenCL
gcc -o ./build/cgmtest_image -I/home/jdegges/local/include -L/home/jdegges/local/lib -
I. cgmtest_image.c -lOpenCL
gcc -o ./build/cgmtest_cached -I/home/jdegges/local/include -L/home/jdegges/local/lib -
I. cgmtest_cached.c -lOpenCL
Done.

Running tests on CPU...
All tests run with key length of 16
Times are averaged over 10 randomized runs
Note that saturated lists are ones with many matches.

============================================

*** Key Maximum: 32 List Size: 100 Saturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: CPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: CPU.

*** Key Maximum: 1000 List Size: 100 Unsaturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: CPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: CPU.

*** Key Maximum: 3276 List Size: 32768 Saturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: CPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: CPU.

*** Key Maximum: 1048576 List Size: 32768 Unsaturated:

[cgmtest.c:141 in gpu_cgm] No CL device found that supports device type: CPU.
[cgmtest_cached.c:493 in gpu_cgm_cacheing] No CL device found that supports device
type: CPU.

============================================

Running tests on GPU...
All tests run with key length of 16
Times are averaged over 10 randomized runs
Note that saturated lists are ones with many matches.

============================================

*** Key Maximum: 32 List Size: 100 Saturated:

Sequential: 0.003600
Binary Search: 0.433500
Simple Parallel: 0.166000
Constant Cacheing: 0.166500
Local Cacheing: 0.201400


*** Key Maximum: 1000 List Size: 100 Unsaturated:
```

```
Sequential: 0.004700
Binary Search: 0.503200
Simple Parallel: 0.165100
Constant Cacheing: 0.168100
Local Cacheing: 0.202200

*** Key Maximum: 3276 List Size: 32768 Saturated:

Sequential: 0.005100
Binary Search: 0.439700
Simple Parallel: 0.165500
Constant Cacheing: 0.173000
Local Cacheing: 0.209900

*** Key Maximum: 1048576 List Size: 32768 Unsaturated:

Sequential: 0.003500
Binary Search: 0.512200
Simple Parallel: 0.167500
Constant Cacheing: 0.165600
Local Cacheing: 0.201700
```