Help Conquer Cancer: Using GPUs to Accelerate Protein Crystallography Image Analysis

Roy Bryant, Adin Scannell, Olga Irzak
Department of Computer Science, University of Toronto

Christian A. Cumbaa
Ontario Cancer Institute

1. Introduction

As part of the greater effort to find a cure for cancer, Igor Jurisica's group at the Ontario Cancer Institute (OCI) is running a project to improve the throughput of protein crystallography [4, 5]. When proteins crystallize, their structure can be determined by observing the refraction of X-rays. Unfortunately, it is difficult to predict under which conditions crystals will form, so proteins are subjected to many automated experiments at various temperatures, pH levels, with various buffers, precipitants, etc. in an effort to form crystals. For a single protein, these automated experiments may involve hundreds or thousands of different solutions; robots periodically capture images of each solution over the course of days or weeks. Currently, the resulting thousands of images (the vast majority of which have no useful information) must be reviewed by human experts in order to determine whether crystals have formed. This process of finding a solution that consistently causes a protein to crystallize is a serious bottleneck and severely restricts the ability of scientists to efficiently study protein structure through x-ray crystallography.

Jurisica's group at OCI has developed analysis software that processes a sequence of images to extract textural features [6]. These features are later fed into machine-learning classifiers to identify experiments that warrant further investigation, i.e. images where crystals may be present. The ultimate goal is to eliminate much of the manual human effort required to find a solution which causes a protein to crystallize. The image analysis software, implemented in C++ by Christian A. Cumbaa, and is currently used by the Help Conquer Cancer (HCC) project that runs on the spare cycles of 250,000 home computers through the World Community Grid [1] using BOINC [2, 3]. This is the same mechanism used by the well known SETI@home project [7].

We have implemented a CUDA-enabled version of the image analysis software that runs significantly faster on computers with CUDA-capable hardware. Speeding up the processing of images is necessary for the project to proceed in a timely fashion. The HCC project was launched in November 2007, and in the first year completed 16% of the total work by processing 24.5 million images in 19,627 CPU years. At the average rate of approximately 7 CPU hours per image, the project is forecast to complete in February 2015. We hope that our faster version of the software will significantly reduce the time required to process the images, and thereby enable cancer researchers to determine the structure of cancer-related proteins faster.

2. Problem

In this Section, we describe the exact nature of the problem and the approach of the existing CPU-based version of the software. As illustrated in Figure 1, images produced in an automated protein crystallography experiment may depict a variety of experimental outcomes. Although the vast majority of images will show nothing of interest, results may include a light phase separation (1(a)), formation of a protein skin at the oil/water boundary interface (1(b)), actual crystallization (1(c)), or simultaneous precipitation and crystallization (1(d)). The automated classification of these images is not trivial, and as a result the image analysis is a complex process.

Feature Extraction

Each image is a digital photomicrograph of a circular, physical well containing the protein and solution that can be easily seen in the examples shown in Figure 1. Basic feature detection algorithms are used to identify the location and bounds of this well within the image, and to define the corresponding region of interest (ROI) within which the analysis is performed.

Although several basic feature extraction algorithms, such as edge detection, are run within the ROI of each image, we find by profiling the image analysis software that the vast majority of time (over 99%) is spent extracting specific textural features [6]. We limited our CUDA-based efforts to the extraction of these features only and henceforth limit our discussion to this. In the evaluation section, we compare CPU- and CUDA-based versions of the software that have all other capabilities disabled.

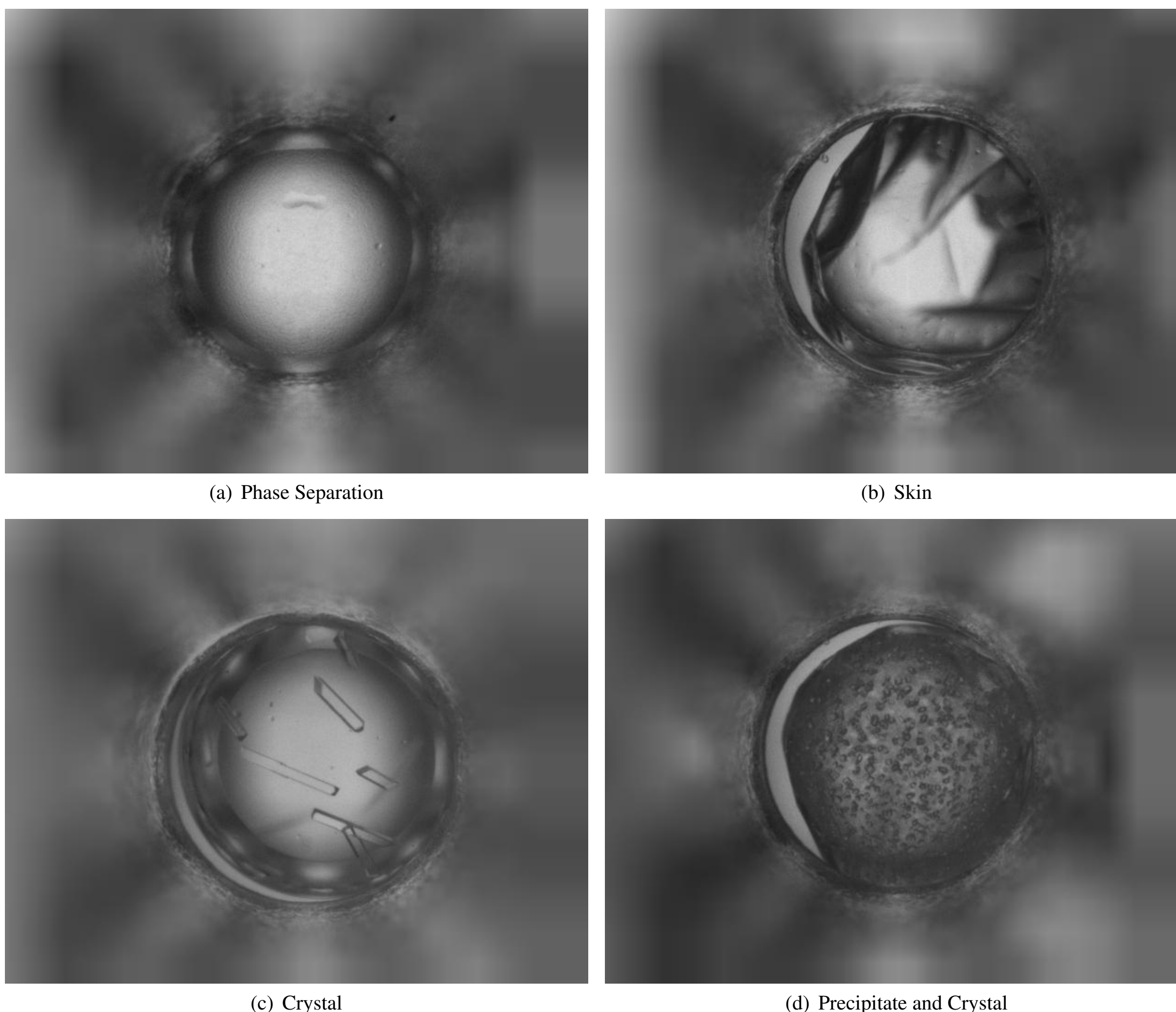
Textural Features

The textural analysis aims to identify the presence of crystals by detecting texture-based features in the image. To do so, it identifies a set of textural features for a large number of Local Windows (LW), then aggregates statistics about those. An LW is the circular region of pixels within a 16-pixel radius, and an LW exists around every pixel at least 16 away from the edge of the ROI.

Given an LW, each set of textural features is defined by two parameters: a distance and an angle. Given these parameters, a grey-level co-occurrence matrix (GLCM) is generated, which is a histogram of values co-occurring at the specified distance and angle. For example, if the distance is 1 and the angle is 0 degrees, then all pairs of pixels with locations $(x, y), (x + 1, y)$ will be counted on the histogram. If pixel values in a 5×1 image are $2, 5, 6, 2, 5$ then the GLCM will have values $(2, 5) = 2, (5, 6) = 1, (6, 2) = 1$ with all other entries 0.

Features are related to the information content of the GLCM, and are extracted once each GLCM has been computed. Intuitively, the GLCM of an image consisting of uniform colours will be heavy along the diagonal, noisy images will result in noisy GLCMs, and images containing a distinct texture will generate GLCMs with significantly more structure hence more information content.

For every distance between one and 25 pixels, statistics are aggregated for all angles within each LW. Then, the statistics from each LW across the entire ROI are aggregated into a single set of values. The statistics of interest are the maximum, mean, minimum



(a) Phase Separation (b) Skin

(c) Crystal (d) Precipitate and Crystal

Figure 1. Several examples of images of protein crystallography experiments.

and ranges of each computed feature value. Thus, for each feature and distance, we produce a maximum, mean, minimum and range.

Furthermore, the above process is done at multiple *quantization* levels. A quantization level corresponds to the width of the GLCM – it is the granularity at which we consider a pixel value. The software currently uses three quantization levels: 64, 32, and 16. For example, at a quantization level of 64, pixel values (at 8-bit grey levels) are divided by 4 before being added to the histogram, which is of size 64×64.

Ultimately, this algorithm considers approximately 1,000 different size, angle combinations for about 22,000 different LWs within the image. With three different quantization levels, the software must generate and process over 66 million GLCMs.

In the CPU-based version, we found that 40% of the time is spent iterating over the image to calculate GLCMs. This process is highly optimized, and GLCMs are computed differentially by adding a new column of pixels and subtracting an old one for each iterated position. The remaining 60% of the CPU time is primarily spent extracting features from the GLCMs. These features vary from simple sums to entropy calculation, but generally involve traversing all elements of each GLCM.

3. Approach

Our approach to porting this algorithm to CUDA was shaped by performance considerations. Initially, we considered porting only one of the GLCM computation or feature extraction phases. However, we determined early on that the cost of copying the GLCMs back and forth between the host and the GPU would be prohibitive. Additionally, extracting the features from each GLCM individually would not perform well due to insufficient opportunities for parallelism. We determined that our design must perform both GLCM computation and feature extraction on the GPU to achieve a significant performance improvement.

We also determined that we would be unable to perform the **entire** analysis within the scope of a single kernel for two reasons: 1) each kernel invocation in CUDA is currently limited to five seconds, 2) the data required by 13 floating point features for 66 million GLCMs is over 3 gigabytes, more memory than most CUDA-capable hardware is equipped with. We identified the computation of all 22,000 LWs simultaneously given a distance, angle combination as a more appropriate parallelizable task. This would require approximately 1 megabyte of memory to store the results (imposing the same in transfer overhead) and approximately 3,000

separate kernel invocations. It also allowed us to use a logical mapping of blocks within a grid to LWs.

Within each block representing an LW, 64 threads are used. These threads are first responsible for reading the image within global memory and computing the GLCM. These same threads then proceed to extract features from the GLCM. The features involve parallelizable operations such as summing GLCM rows, summing the square of values within the GLCM, etc. Finally, the extracted features are written to global memory. When the entire grid has finished, all features are copied back to the host memory.

We refactored the existing analysis code to store temporary statistics as needed and iterated over all possible distance, angle combinations, invoking our kernel once each time. The image and masks are copied to GPU memory only once, before the first kernel invocation. Similarly, memory for the results are allocated only once, however the intermediate results are copied back and integrated into the intermediate statistics after each kernel invocation.

4. Optimizations

In this Section, we briefly discuss some of the optimizations we introduced to our basic design.

Use of Constant Memory

A masks is used to efficiently clip accesses to the circular LW. Since the elements of the masks are accessed in a non-linear fashion, performance is poor when the masks are stored in global memory because the reads are not coalesced. Storing the masks in constant memory benefits performance since the LW mask is referenced very often, and with a size of only 36 pixels square is small enough to entirely fit within cache of modern CUDA hardware.

Coalescing Global Memory Access

Although the image remains constant and is referenced extensively by all threads, it is not a good candidate for storage in constant memory since it is large enough that the frequent access of random pixels would likely churn the cache and reduce performance. Furthermore, each block reads a slightly different portion of the image – unlike masks which are shared by all blocks. We instead store the image in global memory and ensure that the thread access pattern allows reads to be fully coalesced. In generating the GLCM, each thread traverses a column of pixels in the LW, and so each half warp simultaneously accesses 16 consecutive bytes of global memory.

The access pattern for the ROI mask, also stored in global memory, exactly matches that of the image. The third object stored in global memory is the array of 13 features extracted from each GLCM. When the features array is output at the conclusion of the computation, it is written in parallel by 13 threads, and so all global memory accesses are coalesced for speed.

In summary, the image and ROI mask are read from global memory and the extracted features are written to global memory. The LW mask is stored in constant memory, and all other data are stored in shared memory.

Use of Shared Memory for GLCMs

GLCMs are accessed frequently and shared within a block, and so are stored in the shared memory region for performance. Each GLCM stores up to 4k elements (64×64), each with a possible maximum value of approximately 1,000. Since avoiding bank conflicts in shared memory is required for optimal performance, we chose to store the GLCM as a row major buffer of two-byte integers. In iterating over the GLCM, each thread traverses a column with a thread for each element in the row, minimizing bank conflicts. In cases where the original sequential code iterated over rows, we exploit the symmetry of GLCMs and perform column-wise traversal for fewer bank conflicts and better performance.

As with all histograms, access to the GLCM during generation is driven by the data, which leads to contention issues in a multi-threaded implementation. The common technique for avoiding contention when building histograms is for threads to store their own sub-histograms that are later merged. This is impractical for our situation, since with 4k possible values the sub-GLCMs would need to be stored in global memory and the storage requirements would be sufficiently large that the opportunity for parallelism would be greatly reduced. Instead, we chose to use atomic operations to address the issue of contention. Since CUDA's atomicAdd operates only on 32-bit integers, we operated on pairs of 16-bit values with the following transformation:

```
#define ATOMIC_INC(glcm, index) \
    atomicAdd((int*)&(glcm[index & 0xfffe]), \
              (0x00000001 << (((index & 1) << 4))))
```

On-the-fly Normalization

To improve performance by reducing the total number of calculations, the original sequential code stores a copy of the GLCM with normalized values. Since the largest GLCMs have 4k elements and shared memory is a scarce resource, we chose in the CUDA implementation to normalize the GLCM on the fly. While this increases the total number of divide operations, it reduces the memory footprint thereby allowing greater parallelism which more than offsets the penalty. Where possible, the normalize operation is pulled outside loops for better performance.

Template Use

Most of the shared memory buffers need to be large enough to hold a row or column of the GLCM. Since shared memory arrays must be specified at compile time, templates were used to scale the buffers to match the size of the GLCM to be computed (16, 32 or 64 elements). This improves the performance for small- and medium-size GLCMs by reducing the footprint in shared memory, thereby allowing greater concurrent scheduling. By running more blocks in parallel on an SM, the latency of memory accesses can be more effectively hidden. For large (64×64) GLCMs, we were unable to reduce the resource footprint to less than half the available resources, and so in this case only a single block can be run on an SM at a time. Reducing the resource footprint to allow several large GLCMs to be processed concurrently is the most promising opportunity for future optimization.

Similarly, the code often computes arrays of intermediate values from the GLCM, and often needs to sum these vectors for use in later calculations. Templates are used to optimize this performance-critical code, with templates that unroll the code for each of the common vector lengths: 16, 32, 64 and 128.

5. Evaluation

Since we are porting well-tested, fully-correct and heavily-optimized C++ code, we use the existing CPU-based version as a gold standard against which we compare our CUDA-enabled implementation. We evaluated our CUDA-enabled implementation and the CPU-based version using a Dell Core 2 Quad 2.6 GHz system with 8 GB of memory, equipped with a nVidia GTX 280. For evaluation, we selected a set of approximately thirty representative images (four of which are shown in Figure 1) and analyzed them with both the CPU- and CUDA-based versions of the software. In both versions, image analysis other than the extraction of textural features is disabled. The additional runtime required by the disabled code is on the order of minutes.

Speed Improvements

The primary goal in this project was to speed up the analysis of images. Analysis using the optimized CPU-based version on our evaluation system took, on average, 134 minutes per image, and so the entire test set took just under three full days to run. Using our CUDA-based version of the software, the average analysis time per image fell to approximately 6 minutes, with the full analysis time reduced to approximately 3 hours for the 30-image set. In both the CPU- and CUDA-based versions, we saw that some images took slightly longer than others, but the standard deviation was proportional to the mean time to completion. These results are shown in Figure 2, with error bars representing the standard deviation over the 30 images.

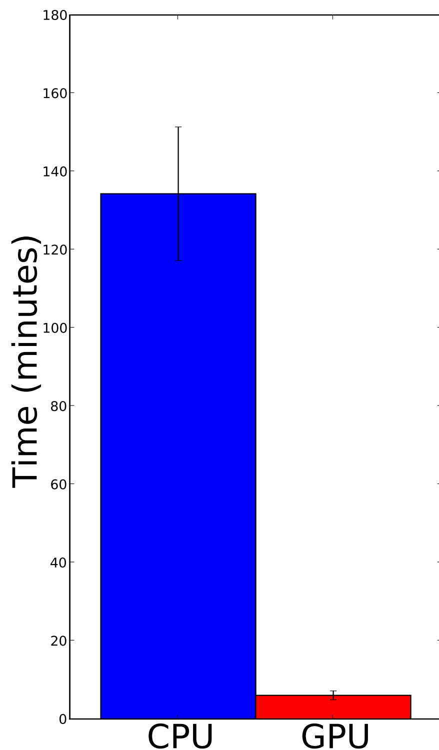


Figure 2. Performance of our CUDA-based implementation versus the optimized CPU-based implementation.

Correctness

Any speedups would not be particularly useful without maintaining the correctness of the feature extraction. Since we use several built-in functions (such as `log`) that the GPU performs at reduced precision for higher performance, we evaluated the stability of the final statistics produced by our implementation. We compare these against those produced by the CPU-based version, which we assume is correct.

Figure 3 shows a cumulative distribution function of the error of the CUDA-based results relative to the CPU-based version. For a value CPU produced by the CPU-based version, and a value GPU produced by the CUDA-based version, the relative error is defined by $|\frac{GPU-CPU}{CPU}|$. If this error is greater than 1, then B shares no significant digits with A. If it is less than one but greater than 0.1,

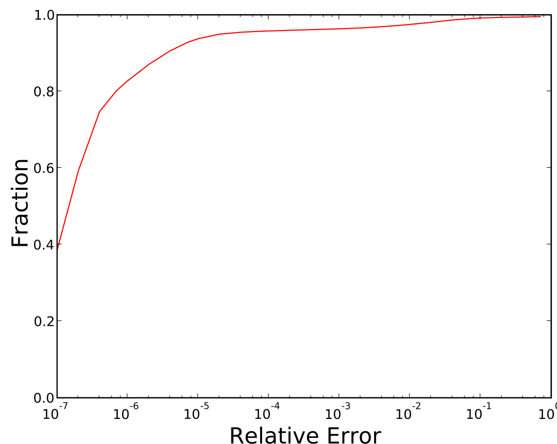


Figure 3. Cumulative distribution of relative error compared to CPU-based implementation.

then B shares only one significant digit with A. Two significant digits correspond to a relative error value of 0.01, etc.

We see that over 95% of statistics are correct to over four significant digits. The larger relative error in the rest of the values are due to errors propagated from the `log` and `exp` functions when they have real values very close to 0. We do not yet have a reason to suspect that these will be problematic.

Analysis of Runtime

After evaluating the performance and correctness of our implementation, we chose to perform fine-grain timing measurements on a set of reference images. We broke down the time spent by our implementation in different components, shown in Figure 4.

We observe that the relationship between the GLCM building and feature extraction components closely mirrors that of the optimized CPU-based implementation, which we profiled but do not discuss here. Since we feel that the most obvious optimizations have been applied in both cases, we find that there are no glaring inefficiencies.

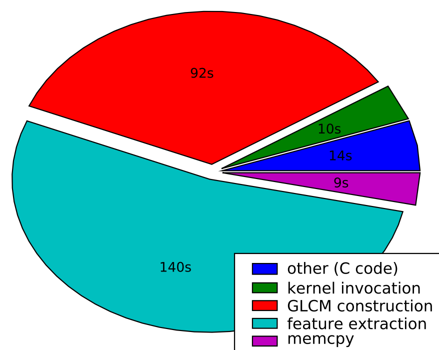


Figure 4. Breaktime of runtime for CUDA-based analysis of an image.

We also note that our design does not impose serious overhead on the CPU, either in terms of copying memory or invoking kernels.

This validates our decision to limit the amount of memory copied from the CPU and the number of kernel invocations. This is an important observation: since the software is designed to be deployed on the World Community Grid, a user may wish to continue to use their CPU while the analysis is running. They are more likely to continue using the software if little load is placed on the CPU.

6. Conclusions

We have successfully built and evaluated a CUDA-based version of the Help Conquer Cancer application. This application extracts features from images of protein crystallization experiments. Only a small amount of work is required to integrate this code into the BOINC client and deploy it onto the World Community Grid.

Our CUDA-based version of the application achieves a significant speed-up over the optimized CPU-based version. We saw a mean performance increase of over 22 times on a set of sample images, without a material reduction in the accuracy of the results. We believe that although the optimizations we applied were rudimentary, our fundamental design is sound and handily realizes most of the possible speed-up without excessive complexity.

References

[1] World community grid. http://www.worldcommunitygrid.org/.

[2] David P. Anderson. Public computing: reconnecting people to science. *Conference on shared knowledge and the web*, 2003. http://boinc.berkeley.edu/.

[3] David P. Anderson. Boinc: A system for public-resource computing and storage. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, p.4-10*, 2004. http://boinc.berkeley.edu/.

[4] Christian A. Cumbaa and Igor Jurisica. Automatic classification and pattern discovery in high-throughput protein crystallization trials. *Journal of structural and functional genomics, Volume 6, Numbers 2-3, p. 195-202*, 2005.

[5] Christian A. Cumbaa and Igor Jurisica. Crystallization image analysis on the world community grid. *Protein Structure Initiative "Bottlenecks" Workshop, Bethesda, MD*, 2008.

[6] Robert M. Haralick, K. Shanmugam, and Its'hak Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics. Vol. SMC-3, pp. 610-621*, 1973.

[7] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. Seti@home - massively distributed computing for seti. *Computing in science and engineering*, 2001. http://setiathome.ssl.berkeley.edu/.