High-throughput protein crystallization on the World Community Grid and the GPU

Yulia Kotseruba[1], **Christian A Cumbaa**[1] **and Igor Jurisica**[1,2]

[1]IBM Life Sciences Discovery Centre, Ontario Cancer Institute, University Health Network, 101 College Street, Toronto, Ontario M5G 1L7, Canada

[2]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada

E-mail: `juris@ai.utoronto.ca`

Abstract.

We have developed CPU and GPU versions of an automated image analysis and classification system for protein crystallization trial images from the Hauptman Woodward Institute's High-Throughput Screening lab. The analysis step computes 12,375 numerical features per image. Using these features, we have trained a classifier that distinguishes 11 different crystallization outcomes, recognizing 80% of all crystals, 94% of clear drops, 94% of precipitates. The computing requirements for this analysis system are large. The complete HWI archive of 120 million images is being processed by the donated CPU cycles on World Community Grid, with a GPU phase launching in early 2012. The main computational burden of the analysis is the measure of textural (GLCM) features within the image at multiple neighbourhoods, distances, and at multiple greyscale intensity resolutions. CPU runtime averages 4,092 seconds (single threaded) on an Intel Xeon, but only 65 seconds on an NVIDIA Tesla C2050. We report on the process of adapting the C++ code to OpenCL, optimized for multiple platforms.

1. Introduction

X-ray crystallography is one of two principal methods of determining the 3-dimensional atomic structure of proteins. Structural genomics, the industrial-scale effort to solve protein structure, is a complex, expensive, and failure-prone process with several bottlenecks limiting the success rate. Protein crystallization is one. Proteins must be crystallized before X-ray diffraction can take place. The chemical conditions for successful protein crystallization are specific to each protein, and difficult to find. Crystallographers must search through chemical space for a *cocktail* (unique combination of buffers, precipitants, and salts) that will crystallize the target protein.

Robotic systems at the Hauptman-Woodward Institute (HWI) screen 1,536 cocktails in parallel against each protein. Each protein-cocktail trial mixes 200 nL of protein solution with 200 nL of cocktail in a conical plastic well. The resulting 400 nL droplet is photographed at multiple time-points, requiring visual assessment by a human expert to determine the outcome of the trial. The High-Throughput Screening lab at HWI[1] maintains and expands an archive of 120 million crystallization-trial images: 13,000 proteins $\times$ 1,536 cocktails per protein $\times$ multiple time-points. Fewer than 1% of these images have been scored by a human expert.

Our goals are to develop an automatic image-scoring system that replaces or substantially assists human experts, and to process the complete HWI image archive. A complete library of

[1] `http://www.hwi.buffalo.edu/faculty_research/crystallization.html`

scored crystallization results will accelerate the search phase in protein crystallization, and will improve throughput, consistency, and objectivity of the results. It will also enable the statistical optimization of the crystallization process, the rational design of crystallization screens, and the discovery of protein-crystallization principles and optimization strategies by mining a database of 20 million crystallization trials.

Section 2 describes our image analysis and classification system, its deployment on the World Community Grid to process the complete HWI archive, and some preliminary results on modeling protein crystallization. Section 3 describes our efforts to port our image analysis system to the GPU.

2. Automated analysis of protein crystallization-trial images

To interpret our images, we first transform each image from raw pixels into a set of numeric features measuring aspects of the image's texture and geometry. A separate classification stage interprets the image based on the features.

We compute 12,375 features per image in a complex, multi-layered, and CPU-intensive image-analysis step. An additional 2,533 features are derived from the primary 12,375, augmenting the feature-set and creating a final set of 14,908 features. All analysis concentrates on a 200-pixel-diameter region of interest centered on the circular droplet in the image. Several categories of image features are measured: basic image statistics, foreground-background topological measures, microcrystal-recognition filters, features measuring local grey-level intensity changes, statistics on the textural and geometric properties of foreground objects revealed by multiple filters (Sobel, Laplacian, and pseudo-Radon transforms), and Haralick grey-level co-occurrence matrix (GLCM) features [1]. Details of the feature computation are described in more detail in previous work [2].

To train and test our image-scoring system, we rely on two large databases of images hand-scored by crystallographers at HWI [3; 4]. In both sets, experts scored each image for the presence or absence of seven crystallization outcomes: *clear droplet*, *phase separation*, *precipitate*, *skin*, *crystal*, *junk*, and *unsure*. These outcomes can co-occur in the same image, with the exception of *clear*, which is exclusive. The first dataset contains 147,456 images, representing 1,536 trials of 96 different proteins, each photographed at a single time-point. Each image from this set was scored independently by at least three experts. The second database contains 17,895 images with the *crystal* outcome, selected from several hundred proteins. Images from this set were scored by a single expert. The two datasets were combined, and a random 10% reserved as a validation set.

2.1. Image analysis on the World Community Grid

The computing requirements for this analysis system are large. The main computational burden of the analysis is the set of GLCM features. GLCM features are computed for multiple neighbourhoods within each image, for multiple direction vectors and distances, and at multiple greyscale intensity resolutions. A total of 13 features $\times$ 68 million GLCMs are computed per image, requiring 4,091 seconds of CPU time on average on a modern PC workstation.

This expensive image analysis is largely being computed on the World Community Grid, a global philanthropic, scientific computing platform powered by the donated CPU-time of its members. Our *Help Conquer Cancer* project launched on the Grid in November 2007. Members have contributed over 75 CPU-millenia to the project as of July 5, 2011, at a rate of 55 CPU-years per day.

2.2. Scoring the crystallization trials

Using our training set of expert-scored images, and features computed from the World Community Grid, we have built a Random-Forest [5] image classifier that distinguishes 11 crystallization outcomes: *clear*, *precipitate*, *crystal*, *phase*, *precipitate & crystal*, *precipitate & skin*, *phase separation & crystal*, *phase separation & precipitate*, *skin*, *junk*, and *other*. Our classifier recognizes 80% of crystal-bearing images (aggregated), 94% of clears, and 94% of precipitates. Table 1 shows the accuracy of the classifier as measured against the independent validation set.

Table 1. Accuracy of the 11-way classifier.

outcome	precision	recall
clear	0.937	0.937
precipitate	0.94	0.717
crystal	0.637	0.622
phase separation	0.653	0.79
precip & crystal	0.422	0.568
precip & skin	0.511	0.796
phase & crystal	0.38	0.348
phase & precip	0.5	0.022
skin	0.545	0.58
junk	0.309	0.769
other	0.269	0.369

2.3. Modeling protein crystallization across time and chemical-space
We are developing a model of protein crystallization that incorporates the changing crystallization state of a single trial over time, and the network of probabilistic relationships linking individual crystallization trials to their neighbours in cocktail space and protein space. In a single crystallization trial, the crystallization state of a trial at time t is related probabilistically to the states at times $t-1$ and $t+1$. Thus, all images in a time series are best interpreted in tandem, effectively embedding the probabilistic outputs of the image classifier in a hidden Markov model. Depending on the starting conditions of the crystallization trial (e.g., supersaturated protein solution), several paths through the various crystallization outcomes are possible. These starting conditions then imply a mixture of HMMs.

Further, two crystallization trials of the same protein with similar cocktails, or similar proteins with the same cocktail, may have similar crystallization reactions. Thus, the outcome of one crystallization trial is probabilistically related to the outcomes of its neighbours in protein- and cocktail-space. Altogether, these layers of probabilistic interconnection form the probabilistic graphical model illustrated in Figure 1.

3. Image analysis on the GPU
Given recent successes elsewhere in applying GPU to multiple problems in image/video processing, bioinformatics, and medical imaging, etc., with reported run-time improvement factors of 10–1000×, we have ported our image analysis system to the GPU. We focussed our efforts on optimizing the GLCM features, which consume 98% of the run-time on the CPU version. We chose the OpenCL 1.0 language due to its support for multiple vendors and device types. We also implemented a CUDA version of our algorithm; however, with the NVIDIA drivers used (270.41.19), we observed no performance gain with CUDA on NVIDIA hardware, and thus kept only the OpenCL version.

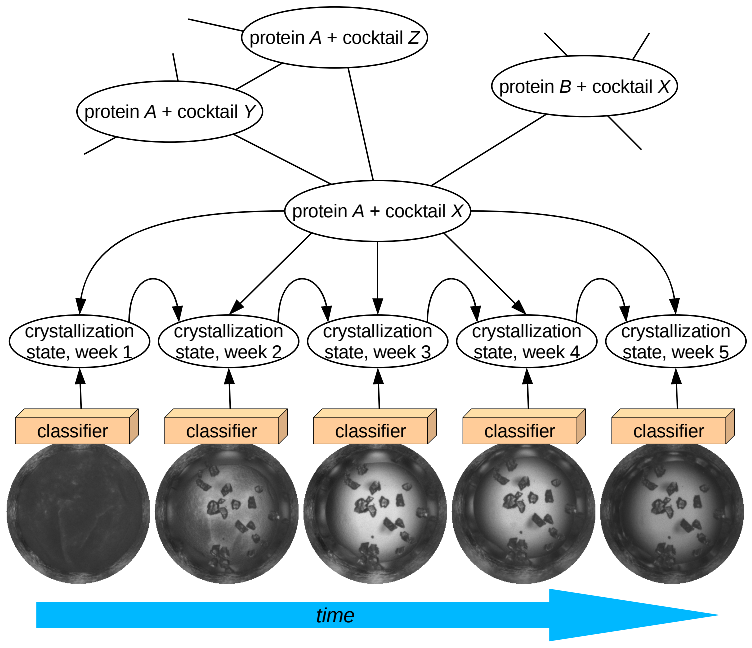
protein A + cocktail Z
protein A + cocktail Y
protein B + cocktail X
protein A + cocktail X
crystallization state, week 1
crystallization state, week 2
crystallization state, week 3
crystallization state, week 4
crystallization state, week 5
classifier
classifier
classifier
classifier
classifier
time

Figure 1. Scheme for a probabilistic model of protein crystallization.

3.1. Implementation Details

The first step in extracting GLCM features from the image is computing the GLCM itself. Every element (i, j) in this 2D histogram represents the co-occurrence frequency of grey levels i and j of pixel-pairs separated by a given (x, y) displacement vector. The width of the histogram is defined by greyscale-quantization level (64, 32, or 16 greys). The circular region of interest of each image contains 22,305 pixels. For each of these pixels, the GLCM is computed in a local window for 1,026 distance/direction combinations and 3 quantization levels, generating over 68 million histograms in total. Each matrix is then used to compute an array of 13 textural features. To reduce the amount of data retained, feature values are aggregated by displacement-vector magnitude for every quantization level. Then for every feature, the maximum range, minimum range, and mean value are stored.

Input image dimensions are 256×256 pixels, so we organize threads in a 256×256 grid with 64–256 threads per block, depending on the hardware. Each kernel computes all GLCMs for a single distance/direction pair, plus all associated features, resulting in 1,026 kernel calls in total. The kernel program is organized as follows: (1) construct the initial 64×64 GLCM from image data in global memory and reduce number of threads to 64; (2) compute the first set of Haralick features; (3) reduce the GLCM to 32×32 and reduce the number of threads to 32; (4) compute second set of features; (5) reduce the GLCM to 16×16 and reduce number of threads to 16; (6) compute the third set of features; (7) write results back to global memory.

This approach minimizes global memory usage: in each kernel execution, the image is read once from global memory for histogram construction, and in the end all features are written into the output array. It also reduces the amount of memory transferred between GPU and CPU: the input, composed of the image itself and several parameters, 65 kB total, and the output arrays, under 3 MB. As a result, the total memory transfer time accounts for less than 2% of the run-time.

3.2. Optimizations
The algorithm depends on statistics computed by summing across all rows or columns of the GLCM. Most GLCMs are sparse, with 10% or fewer non-zero values, usually tightly grouped along the main diagonal. It is not unusual also for a matrix to have a single non-zero value. We used sparseness and the symmetric property of GLCMs to limit iterations to non-empty rows and to simplify computation of some of the features.

Besides the GLCM, most Haralick features depend on one another, and on additional statistics. We cached these values to avoid expensive re-computations.

We limited the use of expensive operations such as `log()`, `exp()` and division. In places where avoiding `log` and `exp` was not possible, we used reduced-precision, hardware optimized functions `native_log()` and `native_exp()` instead. Division is not supported on GPUs directly and is normally implemented as reciprocal and multiplication. We were able to eliminate almost all division operations by precomputing reciprocals on CPU. These arithmetic optimizations alone made our algorithm run 13% faster. Conditional statements were replaced by `select()` function calls. Loops were unrolled where possible. Where applicable, we provided additional compiler options. The NVIDIA compiler tends to allocate more registers than needed, so we used `-cl-nv-maxrregcount=20` to limit register use per thread, and `-cl-mad-enable` for reduced-precision multiply-add operations, helping to speed up index calculations.

3.3. Problems
3.3.1. OpenCL implementations and hardware differences. We have dealt with NVIDIA and ATI implementations for Linux and Mac OS. Theoretically, a program written to the OpenCL API specifications will run correctly on all supporting devices. Obviously, this does not guarantee optimal performance due to the differences in hardware architectures. We intend to run our application on a variety of devices on all major platforms, so we wrote separate versions of the kernel code for CPU, NVIDIA and AMD cards. Note that for Intel CPUs on Mac OS 10.6, OpenCL always runs in single-threaded mode, and the ATI Stream SDK allows for up to 1,024 threads per block in a 1,024 $\times$ 1,024 $\times$ 1,024 grid on a CPU. We noticed that our application does not run efficiently on video cards with no dedicated memory, including almost all mobile GPUs. In some cases, the GPU version runs slower than CPU code. Further, some operating systems impose kernel run-time limits, usually 5–10 seconds. If a kernel exceeds this time, it is killed by the driver. To prevent this from happening, we run the OpenCL kernel on the CPU, which yields over 4-fold performance improvement compared to the original C++ program.

We encountered inconsistencies between NVIDIA and ATI implementations of OpenCL. For instance, memory barrier functions are treated differently. On ATI, every thread must hit the barrier before the kernel returns, whereas on NVIDIA every *active* thread must hit the barrier. So having a thread return before the `barrier()` on ATI will stop the execution indefinitely, while on NVIDIA it runs without any problems. Other minor differences between NVIDIA and ATI compilers changed across several driver updates.

3.3.2. Shared Memory Usage. One of the bottlenecks of this implementation is high usage of shared memory: almost 10 kB are required for each block to store GLCM arrays (8 kB), features, and intermediate results (2 kB). Most of the consumer cards and mobile GPUs have 16 kB of shared memory available. Some of the recent ATI mid-range cards have up to 32 kB and only top computing and gaming GPUs have 48 kB. As a result we could run at most 1–4 blocks of threads, while top cards usually allow for up to 8 blocks running concurrently. This essentially means that we were using only about half of the resources available (the NVIDIA profiler showed a maximum load of 66%). The only way to reduce the shared memory footprint was to compute GLCMs and features in two separate kernels, with the first kernel still requiring 8 kB, and the second much less. Since the GLCM computation consumes 50% of the kernel time,

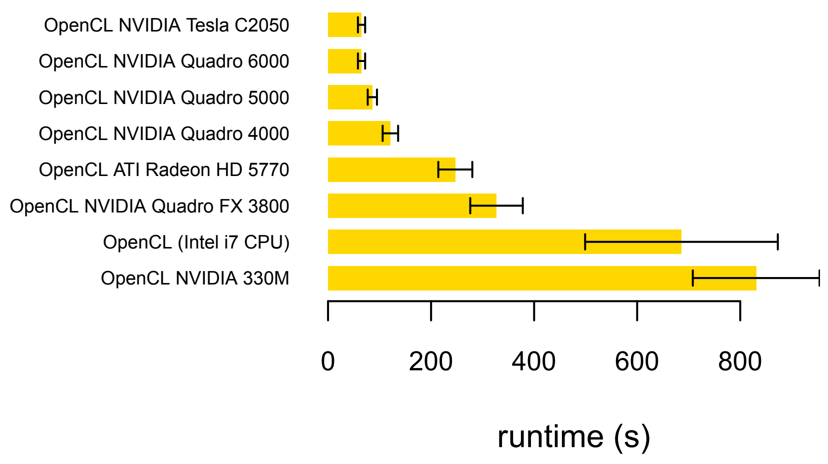

Figure 2. Image analysis run-times for OpenCL on NVIDIA (GPU), ATI (GPU), and Intel (CPU) platforms. The original, optimized C++ run-time is 4,092 seconds (not shown).

some performance gains were expected. In practice, no gains were realized, due to the costs of CPU↔GPU matrix-data transfers between kernel calls, and frequent access to the GLCMs in global memory during feature computation. Additionally, the GLCMs occupied 180 MB of space (22,305 pixels $\times$ 64 $\times$ 64 $\times$ 2 bytes), and most cards had a limit of 128 MB for a single global memory allocation.

3.4. Evaluation

Our main goal in this project was to improve performance without sacrificing precision. To evaluate our OpenCL implementation, we used as a standard the existing optimized C++ code, as runs on the World Community Grid. We ran our OpenCL implementation on several NVIDIA cards ranging from mobile devices to the newest scientific computing cards.

3.4.1. Performance Improvements. We were able to achieve a significant performance improvement for this problem. Figure 2 shows the time required to compute the Haralick features, averaged over 50 images randomly selected from our test set. Our final version of OpenCL code runs 6–60 times faster than the original optimized C++ code. It took over 3 days of CPU time to process our small 50 image test set, and under 2 hours on the NVIDIA Tesla C2050 GPU.

Kernel time is almost equally split between computing histograms and features. This ratio differs from the CPU version, which spends about 70% of time computing features. More detailed breakdown of GPU runtime is given in Figure 3.

We also measured the GFLOPS for our GPU implementation. On the Tesla C2050 we achieved 5.5 GFLOPS, which is only 1% of its maximum performance of 515 GFLOPS. Such low efficiency could be caused by several factors. None of the integer operations contributed to GFLOPS, so array-indexing and histogram-building was not taken into account. Additionally, the Haralick features require very complex memory access patterns and there were insufficient arithmetic operations to compensate for memory latencies and bank conflicts. Finally, as

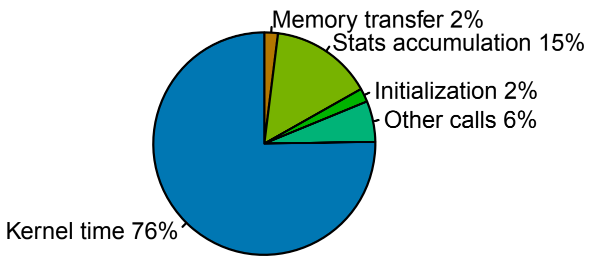

Figure 3. Breakdown of GPU run-time.

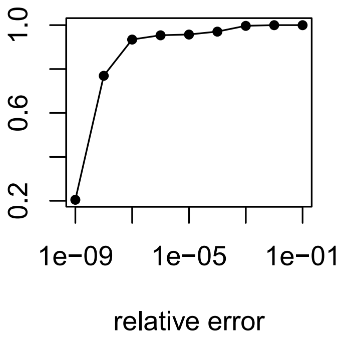

Figure 4. Cumulative distribution of feature values exceeding the relative error threshold.

described above, we were unable to saturate the GPU pipeline due to the high shared memory usage and gradual reduction of number of active threads.

3.4.2. Correctness. There is always a trade-off between precision and performance in GPU. Most current graphics hardware is not fully compliant with IEEE 754 standard for single-precision floating point operations. In particular, NVIDIA video cards flush denormals to zero and do not guarantee full precision for division and square root operations [6]. Since loss of precision is inherent in GPU computation we needed to measure its effects on computed results. We first computed relative error of OpenCL versus CPU results, defined as $\left|\frac{GPU-CPU}{CPU}\right|$. In the $CPU = 0$ case, we took an absolute error. It follows from the cumulative distribution plot on Figure 4 that almost 97% of values are correct up to four significant digits. More important is any possible effect on image classification. To verify that any shortcuts we took in the OpenCL version would not affect our classification, we tested the final code on a 4,500-image subset of our image-classifier's training/testing set. We then applied our classifier to both GPU- and CPU-computed image features and found only eight images that were assigned different classes. All eight of these images belonged to an ambiguous category (human experts disagreed on the image's true class), suggesting that discrepancies between GPU and CPU features pose little problem.

4. Conclusion
We have built and tested GPU-accelerated version of optimized C++ code. Given the complexity of the problem, we still achieved a 6–62-fold performance increase, depending on the hardware and OS. Despite some loss of precision, features computed on the GPU still give classification results consistent with the gold-standard CPU version. The GPU version therefore runs fast enough to allow same-day analysis of crystallization-trial images, and correctly enough to run alongside the CPU version on the World Community Grid. We expect GPU-enabled *Help Conquer Cancer* to enter beta testing by the end of 2011.

Acknowledgments
We wish to thank and acknowledge IBM and the World Community Grid volunteers for providing free computational power and technical support for this research through World Community Grid. For further information about World Community Grid, please view the following: `https://secure.worldcommunitygrid.org/index.jsp`. This work would not be possible without data generated at Hauptman-Woodward Medical Research Institute and insight from many discussions during our collaboration with Drs. George T. DeTitta, Edward H. Snell, Joseph R. Luft and their team, Geoff Franks, Thomas Grant, Dr. Stacey M. Gulde, Mary Koszelak-Rosenblum, Angela M. Lauricella, Dr. Michael G. Malkowski, Raymond Nagel, Stephen A. Potter, Meriem I. Said, Jennifer L. Smith, Max Thayer, and Christina K. Veatch. Dr. Jerry Heyman and Miso Cilimdzic from IBM assisted with testing, profiling, and tuning the CPU version of our image analysis application. Roy Bryant, Adin Scannell, and Olga Irzak at the University of Toronto developed an early CUDA prototype. We wish to thank Lenovo and NVIDIA for providing PC and GPU hardware for development and testing, and Richard Lu for assisting with computing resources and environment for post-Grid data processing. The authors gratefully acknowledge funding by NIH U54 GM074899, the Canada Foundation for Innovation (Grants #12301 and #203383), the Canada Research Chair Program, the Natural Science & Engineering Research Council of Canada (Grant #104105), the Ontario Research Fund (GL2-01-030), and IBM.

References
[1] Haralick R M, Shanmugan K and Dinstein I 1973 *IEEE Trans. Syst. Man Cybern.* SMC **3** 610

[2] Cumbaa C A and Jurisica I 2010 *J. Struct. Funct. Genomics* **11** 61–69

[3] Snell E H, Luft J R, Potter S A, Lauricella A M, Gulde S M, Malkowski M G, Koszelak-Rosenblum M, Said M I, Smith J L, Veatch C K, Collins R J, Franks G, Thayer M, Cumbaa C A, Jurisica I and Detitta G T 2008 *Acta Crystallogr.* D **64** 1123

[4] Snell E H, Lauricella A M, Potter S A, Luft J R, Gulde S M, Collins R J, Franks G, Malkowski M G, Cumbaa C A, Jurisica I and DeTitta G T 2008 *Acta Crystallogr.* D **64** 1131

[5] Breiman L 2001 *Mach. Learn.* **45**(1) 5–32

[6] Whitehead N and Fit-Florea A 2011 Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs Tech. rep. NVIDIA Corporation URL `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`