Memory Constrained Iterative Phase Retrieval Using GPGPU

Ladislav Mikeš

Institute of Computer Science, Faculty of Science
Pavol Jozef Šafárik University in Košice
Košice, Slovakia
Email: ladislav.mikes@upjs.sk

***Abstract*—Prediction of a scattering experiment for the new generation of coherent X-ray sources, such as X-Ray Free-Electron Laser (XFEL) requires a significant advance in both precision and numerical effectivity of radiation damage modeling. The preferred method of analyzing data from such experiment to obtain structure information relies on an iterative approach of refining an object estimate by repeatedly simulating the beam propagation between exiting the object and arriving at the detector. Currently, the most cost-effective way is to delegate work to General-Purpose computing on Graphics Processing Units (GPGPU), working around its limitations given by a reduced instruction set, limited memory size and increased latency when accessing the rest of the system. In this paper, we present managing fast GPU memory when performing reconstruction of an object consisting of up to 128 million voxels on currently available devices with less than 6GB of onboard memory.**

Keywords–Radiation scattering; Inverse problem; Reconstruction; GPGPU.

I. INTRODUCTION

The emerging of new generation of coherent X-ray sources is characterized by high fluence, femtosecond scale flashes and high repetition rate. After vetoing blank and otherwise defective measurements, we expect a stream of up to 1000 frames per second to be further analyzed. High fluence provides us with enough photons to image a single molecule without need to create crystals, but at the cost of the sample getting destroyed in a rather violent explosion. A detector capable of sampling at 4.5MHz coupled with a very short flash lets us measure the sample when the explosion is just starting and its effects are still negligible [1][2], but to retain detail, we have to consider ongoing radiation damage and integrate over multiple time slices. This requires thousands of the forward scattering problem instances to be solved and considering the projected detector resolution of 1 and 4 MPix, the required computational performance is only attainable by massive parallelism and usage of numerical accelerators, such as GPGPU.

In this contribution, we focus on the partial problem of optimizing limited size single particle reconstruction from low-angle scattering pattern, where only a single node equipped with a single GPU is used. We are going to reconstruct wave phase on detector using the iterative phase retrieval method.

While larger samples can be fixed to a motorized holder to undergo a tomographic scan at smaller doses of radiation, single particles at XFEL are injected into the beam in a stream, losing information about orientation. Common practice is to match the orientation of multiple shots, perform 2D phase retrieval and run a tomographic reconstruction. There are, however advantages [3] to first generating diffraction intensity volume and then performing 3D phase retrieval. Due to limited GPU onboard memory, performing 3D phase reconstruction at projected detector resolutions with GPGPU is an intricate task and our ongoing research aims to find a reasonable solution.

In Section 2, we describe the general algorithm for phase reconstruction. Section 3 gives an overview of the GPGPU architecture. Section 4 enumerates memory requirements of the basic phase reconstruction algorithm as discussed in Section 2. In Section 5, improvements to address memory consumption concerns are proposed. Results of a test run are shown in Section 6 and conclusions are presented in Section 7.

II. PHASE RECONSTRUCTION

In a scattering experiment, we examine the object interacting with incident radiation. Finding the object corresponding to measured data is more or less straightforward in optical spectrum, but in X-ray, depending on material and wavelength, the imaginary part of refraction index, causing phase shifts in beam, becomes crucial.

While modern detectors are capable of counting X-ray photons at acceptable rates, phase information is lost. However, the phase information can be inferred providing sufficient oversampling [4]. One of the phase problem solutions is the algorithm called iterative phase retrieval [5]. It works on an object estimate and makes gradual improvements to it based on feedback from scattering simulation. To be able to harness the duality of object and its diffraction pattern, we need to understand how a wave evolves between exiting the object and being measured on a detector.

The easiest solution is to place the detector relatively far from our sample and use small angle scattering, which numerically translates to using Fourier transforms, in our case Fast Fourier Transform (FFT). We could also use X-ray optics to further enhance resolution, but this requires more complex propagators and increases reconstruction difficulty.
The general algorithm is as follows:
Start with a flat or a fully random object estimate.

1) Propagate beam interacting with object estimate to detector distance
2) Update simulated amplitudes to reflect measured data, leave phase unaltered
3) Propagate beam back by an inverse operator
4) Update object estimate

We iterate these steps over until the object converges and there is nothing to update in step 2.

Since a detector is usually neither a single chip nor a

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-442-8

borderless composite, blind spots are inevitable. Usage can also destroy individual pixels and we will need to ignore them. We therefore keep a pixel validity mask and update only valid ones in step 2.

We usually start with a random object and the best results can be achieved by simply executing the algorithm multiple times and averaging the results.

The object should only take up a portion of reconstruction volume. We employ a support function to differentiate between object and supposedly empty space. The general idea is to bring object estimate inside support closer to backpropagated values while suppressing areas outside support. The support function can be derived from other experiments such as electron microscopy or just estimated roughly, but reconstruction precision and speed require a well-behaved tight support. Ideally, we could derive the support function from object estimate itself by shrinkwrap technique [6], based on a low-pass filter and thresholding. It is numerically more demanding than other steps of one iteration, but we only need to update the support function once every few iterations.

We could also assume object to be positive, since it cannot absorb more radiation than there is and usually does not spontaneously emit radiation either. This, however, can be counter-productive and leave us unable to find the right solution so we use the positivity constraint in a different form.

There has been significant progress made to improve the step 4 [7] and we are currently using the Relaxed Averaged Alternating Reflections method [8]:

$$object_{n+1} = object_n^P + \beta * (object_n - 2 * object_n^P) + \beta * \phi_{S_+}\left(2 * object_n^P - object_n\right) \quad (1)$$

$$object_n^P = \phi_{BPROP}(\phi_A(\phi_{PROP}(object_n))) \quad (2)$$

Where
$object_n$ is the object estimate in iteration n,
$object_n^P$ is the object estimate backpropagated from step 3
$\beta \in [0, 1]$ is the update factor
operator ϕ_{S_+} applies support and positivity constraint, ignoring (zeroing) negative values
ϕ_A applies measured amplitudes in detector space and
ϕ_{PROP} and ϕ_{BPROP} propagate the exit beam to detector space and back
We further require the information about convergence so we know how much we can trust the reconstructed object. It is extracted as the magnitude of change we had to perform in step 2.

Even though the imaging experiment itself is at most in 2D by design, we are able to run it multiple times for various orientations and put them all together to create a 3D diffraction volume. The same algorithm then applies, but now we use 3D FFT for our propagation instead of its 2D counterpart. For 3D object phase reconstruction we hit memory constraints due to architectural limitations of GPGPU. The ways how to deal with the GPGPU memory constraints are the topic of presented contribution.

III. GPGPU

With increasing demand for 3D features and performance, power of dedicated graphics cards rose rapidly. With higher resolutions, more memory is also required to store color, depth and stencil buffers, as well as high quality textures and other features.

After the introduction of programmable shaders in the early 2000s, enthusiasts learned to repurpose these to perform various computations on the graphics card, taking advantage of their highly parallelized architecture. The first release of general purpose computing on GPU was with the Nvidia 8800 series, the first CUDA-capable card.

There are 2 major platforms for GPGPU : CUDA by Nvidia and OpenCL by consortium Khronos Group.

- CUDA is available only for chips made by Nvidia and is popular for its easy to use API. Device code (an extension of C) is compiled using a specialized tool and linked into the executable.
- OpenCL was created as an open standard and any GPU manufacturer is welcome to support it. Device code (in C) is loaded, compiled and linked at runtime, making it a somewhat more difficult to manage.

A. Construction

A GPGPU device usually consists of a controller, onboard memory and compute units: cuda cores or stream processing units. A compute unit contains a number of computation cores with their private registers and a small amount of locally shared memory. A program is executed on a block of cores, all performing the same instruction.

Massive parallelism is the main advantage of GPGPU. While individually not as powerful as CPU cores, the sheer number of cores available within a single device makes GPGPU the ideal choice for element-wise operations, a major part of phase retrieval algorithm.

When considering GPGPU for scientific research, it is also important to note their double precision (DP) performance and error correction in memory. These features are exclusive to professional grade cards with double precision performance being reduced to half or even eighth of their single precision (SP) performance as opposed to up to $1/32$ in consumer gaming cards. In our case double precision also doubles the memory requirements for most buffers, a price we may not be able to afford in some cases.

B. Memory hierarchy

We can utilize various types of memory on GPU:

- registers are very fast but only in limited supply
- local memory is shared between threads in a block, up to 48kB depending on architecture, 1TB/s throughput
- global memory is shared between everything on device, up to 12GB in current top-end devices, up to 320GB/s
- peer to peer access accessing another device's global memory is limited by speed of their PCIe bus, up to 32GB/s
- system memory limited by PCIe bus and RAM itself, up to 32GB/s

We can see that using memory outside the device is by an order of magnitude slower and is usually used only to load data and save results. Even adding another device with very fast memory of its own may not be a good choice in some cases due to communication bottleneck.

Best practice in cases where memory size is not an issue is to load all input data to GPU memory, perform computation

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-442-8

and save result back to system memory. For problems with localized memory access, we can load only data relevant to current batch being computed and reuse buffers for next batch. There are mechanisms built into current GPGPU architectures to allow for data transfers to/from GPU for next/previous batch while current batch is being computed without significant performance degradation due to concurrent memory access. In our case, we need to have immediate access to all the buffers for various steps of each iteration.

IV. Memory usage

When performing iterative phase retrieval as discussed above, we need to consider not only the data structures needed to store data, but also temporary storage.
We will express memory requirements using $size$ as it mostly depends on reconstructed volume resolution.

A. Basic data structures

As the algorithm suggests, we need place to store:

- input data: $size$ floats
- input validity mask: $size$ booleans, bytes at best
- object estimate: $size$ floats
- object backpropagated: $size$ floats
- support function: $size$ booleans
- average: $size$ floats, [optional]

B. Fast Fourier Transform

There are existing implementations for FFT for both major GPGPU platforms: cuFFT for CUDA and clFFT for OpenCL, both of similar performance. FFT in general requires input and output buffers and temporary storage

- input buffer: $size$ complex floats
- output buffer: $size$ complex floats
- temporary buffer: $size$ complex floats

Commonly used FFT implementations place the zero-frequency element after transforming to the frequency domain at the beginning of buffer, as opposed to the center of image acquired by detectors. While input data can be shifted before some of the other buffers are populated, the shrinkwrap algorithm requires access to zero-centered data or suffers performance losses. Shifting between normal and 0-centered layout requires an additional buffer since it cannot be done with limited register/local memory in-place.

- shift buffer: $size$ floats

C. Sequential operations

During thresholding and error calculation, we may need to find the maximum or sum of a buffer. This is seemingly sequential task, but it can be parallelized quite well using the divide-and-conquer strategy.

With built-in mechanisms for synchronization within a thread block we can compute maximum or sum of some part of a buffer. This could be the whole buffer, but then we are limited to using only one thread block and work is done mostly sequentially. What we can do is to have multiple blocks compute over parts of the buffer and when all are finished, we find the maximum or sum of their results. We only need some additional space to hold partial results. Its size does not depend on reconstruction dimensions, but rather on the number of blocks we wish to use, which is almost negligible.

- divide-and-conquer buffer: $\#blocks$ floats

D. Shrinkwrap

To perform low-pass filtering, we need a buffer to store temporary result before finding the maximum and applying threshold.

- shrinkwrap buffer: $size$ floats

E. Total usage

To sum it all up, we require up to $size * 12$ floats , $size * 2$ bytes and another $\#blocks$ floats. In single precision, this is

$$size * 50 + \#blocks * 4$$

bytes, which for object with 128M voxels (and a negligible number of blocks in comparison) is 6400MB of memory.
The Nvidia TESLA card we were using only had 5.3GB onboard memory, but the memory requirement estimate is upper bound and we can further improve it.

V. Improvements

To fit all the required buffers into available memory, we had to modify the straightforward implementation of iterative phase retrieval and place some limitations on object size.

A. FFT buffers

We can see that the detector space buffer is only used to adjust amplitude. We could keep freeing and reallocating it as needed, but we can also repurpose the space when it is not used.

While CUDA allows us to declare a buffer inside another just by adjusting pointers, in OpenCL we only work with buffer handles and we have to ask for a subbuffer to be created inside another. Another way is to leave one buffer out completely by switching to in-place Fourier transforms.

B. Fourier shift buffer

If reconstruction size is even in all dimensions, shifting is reduced to swapping values in pairs and is possible to do in-place. All we have to do is to pad the input to suitable dimensions, which should be done anyway to ensure good FFT performance.

C. Detector mask

Since the sole purpose of detector mask is to invalidate pixels/voxels, we can do this on the detector data itself. As the amplitude is supposed to be positive, we could use negative values for signify invalid data.

D. Updated total

By doing these simple modifications, we can save $size * 3$ floats and $size$ bytes, reducing total memory usage to

$$size * 37 + \#blocks * 4$$

bytes (SP).

Following mentioned improvements, reconstruction of an object enclosed in a cube with side length of 512 (up to 128M

voxels) now requires 4.74GB of onboard memory for single precision, which fits our device with a safe margin. For double precision it would be 9.34GB. If we wanted to work with data from a 1MPix detector and a cube with 1G voxels, we would need 37GB(SP) or 73GB(DP).

VI. RESULTS

We implemented the algorithm described above to be part of a toolbox called SingFEL as a means of reconstructing objects from experimental data for the Single Particles, clusters and Biomolecules instrument at European XFEL.

The reference implementation on CPU was designed to use matrix element-wise operations, but was only compiled with single-thread support for testing purposes. We compared the runtime for objects of various sizes and the time spent solely on iterations, without I/O or host-device transfers and the GPU version was about 70 times faster.

Shown in Table I is runtime and maximum observed memory usage when running 100 iterations of phase reconstruction. Memory usage increase on CPU can be accredited to Armadillo, linear algebra library, [9] creating temporary buffers for operation results, while in GPU the CUDA runtime requires some additional memory but otherwise fits with our estimates.

TABLE I. PERFORMANCE COMPARISON CPU-1 vs CUDA

Input	*CPU*	*RAM*	*GPU - CUDA*	*VRAM*
128^3	90s	160MB	1.5s	120MB
256^3	843s	1200MB	12s	620MB
512^3	7020s	9000MB	101s	4810MB

The test was performed on a phantom object (Figure 1)

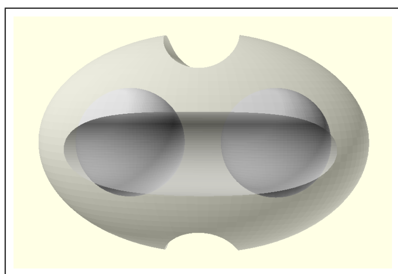

Figure 1. Phantom object

While not representative of our intended target object, a sub-cellular biological sample, it offers well-defined borders and is homogeneous inside for easy error assessment.
To simulate a real detector with blind spots, we applied a tic-tac-toe shaped 3D mask to detector data, accounting for missing detector information (Figure 2)
The result after 250 iterations (Figure 3) shows the correct general shape, but is still noisy. We are losing a significant amount of information by masking pixels out.
The error function (Figure 4) approaches zero as there ceases to be any activity during amplitude update.

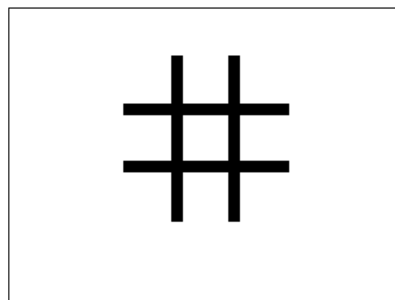

Figure 2. Detector mask, central cut

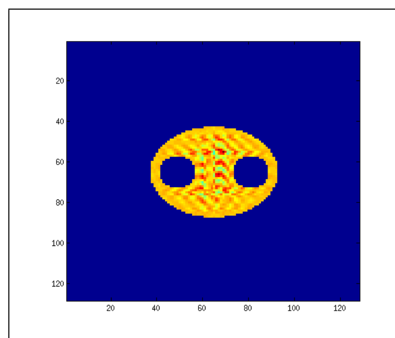

Figure 3. Reconstructed object, frontal cut, off-center

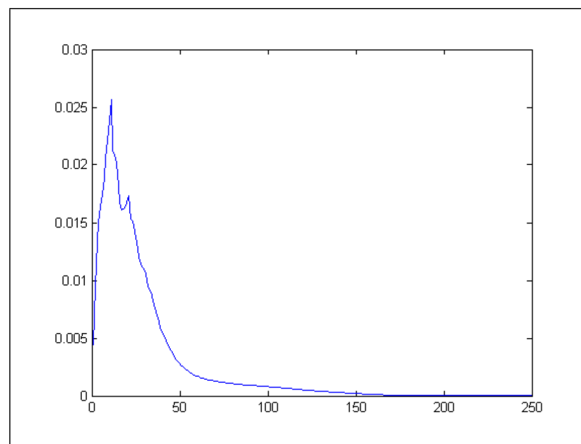

Figure 4. Error measurement

As we can see, the object converges after approximately 170 iterations. If detector data are unmasked, convergence occurs after less than 100 iteration and the difference from input is almost immeasurable in single precision. With real experimental data, the algorithm usually requires hundreds if not thousands of iterations to converge, making the overhead of copying data to GPU insignificant in comparison.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-442-8

VII. Conclusion

We have optimized for GPGPU and implemented the Relaxed Averaged Alternating Reflections method with shrinkwrap-based periodical support update to be included in SingFEL for object reconstruction. We show the best achievable case and and why the 1024^3 resolution derived from intended detector resolution is not possible with current hardware and our single GPU limitation. While this is quite certainly a setback, best usable case is still satisfactory for immediate deployment within SingFEL, a software suite for single particle imaging at FEL (in development).

With an average speed-up of 70 over single-threaded CPU implementation, we have also supported the claims about GPGPU being the most cost-effective solution (for reasonably sized problems) to date, as a system with 70+ CPU cores is not going to scale linearly in performance.

At the time of writing this paper, professional single-GPU cards top out at 16GB and dual-GPU cards at 24GB of onboard memory. The first single-GPU card to have 32GB onboard memory was just announced, but still falls short of the estimated 37GB memory required for 1G voxel reconstruction in single precision. Dual-GPU cards may therefore soon surpass this requirement but are still expected to be just as performance limited as multiple cards by having to communicate through the PCIe bus. To push the resolution further, we have to accept a slight bottleneck in inter-device transfers. One of the possibilities would be to dedicate one card to detector (Fourier) tasks. This way, all (2 or 3) FFT buffers and detector data could reside on one card and everything else on the other one. In each iteration we then need to transfer object estimate forward and backpropagated object estimate back, totaling 8GB transfers for 1G voxel reconstruction, delaying each iteration by approximately 1 second at 12GB/s compared to the estimated 8s for a single iteration were it all performed on a single device. We consider this drawback tolerable and will proceed in this direction further.

With double the memory requirements, we did not pay much attention to double precision and the advantages it might bring. Rounding errors usually propagate through iterations, but in case of iterative phase retrieval, the mechanism of replacing amplitudes with measured data creates a fixed point in our computation and error does not compound itself.

On the opposite side, we have the recent introduction of 16-bit floats with the newest CUDA framework. While cuFFT does not yet support half precision, we are considering its feasibility for further development.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (call FP7-REGPOT-2012-2013-1, proposal 316310) "Fostering Excellence in Multiscale Cell Imaging (CELIM), USP Technicom (ITMS 26220220182) and VVGS-PF-2014-451.

References

[1] R. Neutze, R. Wouts, D. van der Spoel, E. Weckert, and J. Hajdu, "Potential for biomolecular imaging with femtosecond X-ray pulses," Nature, vol. 406, no. 6797, Aug. 2000, pp. 752–757.

[2] Z. Jurek, G. Faigel, and M. Tegze, "Dynamics in a cluster under the influence of intense femtosecond hard X-ray pulses," The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics, vol. 29, no. 2, May 2004, pp. 217–229.

[3] N. D. Loh and V. Elser, "Reconstruction algorithm for single-particle diffraction imaging experiments," Physical Review E, vol. 80, no. 2, Aug. 2009, p. 026705.

[4] R. H. T. Bates, "Fourier phase problems are uniquely solvable in more than one dimension. I: underlying theory," Optik, vol. 61, 1982, pp. 247–262.

[5] R. W. Gerchberg and W. O. Saxton, "A practical algorithm for the determination of the phase from image and diffraction plane pictures," Optik, vol. 35, 1972, pp. 237–246.

[6] S. Marchesini, H. He, H. N. Chapman, S. P. Hau-Riege, A. Noy, M. R. Howells, U. Weierstall, and J. C. H. Spence, "X-ray image reconstruction from a diffraction pattern alone," Phys. Rev. B, vol. 68, Oct 2003, p. 140101(R).

[7] J. R. Fienup, "Phase retrieval algorithms: a comparison," Appl. Opt., vol. 21, no. 15, Aug. 1982, pp. 2758–2769.

[8] D. R. Luke, "Relaxed averaged alternating reflections for diffraction imaging," Inverse Problems, vol. 21, no. 1, 2005, pp. 37–50.

[9] C. Sanderson, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," Technical Report, 2010.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-442-8