GPU Computing in XAFS

K Pedersen and G Bunker
Department of Physics, Illinois Institute of Technology, Chicago, IL

E-Mail: bunker@iit.edu

Abstract: Modern graphics processing units (GPUs) are powerful parallel computers. Porting the mathematically intense, parallelizable algorithms of XAFS to run on GPUs should expand the scale of tractable XAFS calculations. Part of a time-limiting subroutine in FEFF 8.4 (fms) was converted to run on a GPU, showing significant performance gains. With parallel CPUs sharing GPU resources, the runtime of the modules calling fms should accelerate by one or two orders of magnitude versus a single-threaded CPU process (at least for large atomic clusters, ). New or existing XAFS algorithms and programs (e.g. FEFF 9) should show similar acceleration when ported to GPUs, as long as they meet certain criteria (outlined in detail).

1. Introduction to GPU Computing
A GPU is essentially a cluster computer condensed onto a single board. Like a cluster, a GPU (or device) has a number of parallel processing nodes, called streaming multiprocessors (SMs) [1]. Since the SMs all reside on the same board and share the same on-chip global memory space (RAM), GPUs sidestep much of a cluster’s overhead. GPUs are controlled by their computer’s CPU (or host) [2].

The transistors of a GPU are optimized for FLOPS (i.e. math), severely limiting a device’s ability to process control logic (like if-then-else) [2]. Each branch in a GPU control tree executes in series, and every branch (even those bypassed by a given control instance) gets a pass through the SM (this is called divergence). Bypassed branches, while computationally impotent, still eat up SM clock cycles. In order to benefit from GPU conversion, (rule 1) algorithms or subroutines must operate in an almost rote manner on many different inputs, with minimal inline control [2]. Hence the GPU niche is large for/DO loops which fill or map matrices (e.g. matrix operations) [1]. Sections of an algorithm that rely heavily on control statements should be handled by the host, which can then outsource niche GPU sections to the device. When the device is busy the host is not idle; after issuing instructions to the device, the host can immediately return to its own tasks (this is called asynchronous operation).

1.1. Compute Unified Device Architecture (CUDA) and CUDA C
The transistors of Nvidia CUDA devices have the versatility to tackle mathematical tasks not usually required in graphics rendering [1]. Using CUDA C, an extension of C, users can harness the power of their CUDA devices with a simple API. CUDA C is free, well supported, and well documented, but it only works on Nvidia GPUs. An open-source alternative, OpenCL, is supported on GPUs from a number of manufacturers, including Nvidia. This paper focuses solely on CUDA and CUDA C.

To translate an algorithm into CUDA C, it is broken into irreducible sets of instructions called kernels, which the host calls sequentially. Each parallel element of a kernel’s task is carried through its code by a thread. Threads are organized into small, 2D arrays called blocks (1024 threads max). Each block is executed by a single SM, with a device’s multiple SMs executing many blocks concurrently. Blocks are arranged into a 3D array called a grid, which is a kernel’s work order [1]. Based on its location in its 2D block, and its block’s location in the 3D grid, a thread can identify which element in a data structure to handle [1]. To complete a kernel call, blocks are taken from the grid and distributed to the SMs. Since devices do not schedule block execution deterministically, (rule 2) blocks must be able to execute in any order; algorithms with data interdependencies must be broken into autonomous kernels [1]. Also, since neither the device nor the host can directly access each other’s RAM; (rule 3) host memory needed by the device, and device results needed by the host, must be transferred [1].
Device RAM (global memory) is external to SMs, so SM read latency is much slower than SM clock speed [2]. To prevent SM idleness (waiting for global memory calls to return), threads in the same block are bundled into warps (cadres of 32 threads). Via SIMD, all 32 threads simultaneously complete the same kernel instruction until they collectively read from global memory; if they access a contiguous region, this coalesced read forms a single transaction [2]. The SM requests the memory, sets the warp aside, and picks up an idle warp whose memory has arrived and been cached, processing the new warp until it reads global memory. Since each SM can currently house only 8 blocks at a time, (rule 4) blocks should contain as many threads as possible so there are always idle, ready warps [2]. Nonetheless, SM memory bus bandwidth is finite, so (rule 5) global memory traffic should be kept to a minimum. Just like carpooling, coalescing reduces global memory traffic by reducing redundancy; but utilizing several other types of GPU memory (shared, constant, textures, registers) is crucial [1]. Learning how to use these memories effectively is the main challenge of GPU computing [1, 2].

1.2. CUBLAS and CULA
Maintaining their original API, BLAS and LAPACK have been translated into CUDA C (as CUBLAS and CULA, respectively). CUBLAS is distributed free by Nvidia. CULA uses CUBLAS under the hood, and is licensed by EM Photonics, with yearly subscriptions for commercial and academic use. Both libraries can be called on Nvidia devices via CUDA C, or directly from FORTRAN.

2. FEFF 8.4
FEFF 8.4 is software, written in FORTRAN 77 and licensed by the University of Washington, which simulates XAFS spectrum ab initio. To calculate self-consistent atomic potentials, the SCF module is activated, creating a long iterative loop [3]. Activating the FMS module calculates the full multiple scattering, which is also extremely time-consuming. However, using these modules helps generate the best simulated spectra, which are most useful for fitting experimental data [4].

2.1. Subroutine fms
The beating heart of both the SCF and FMS modules is the subroutine fms. For a cluster of \( n \) atoms, fms uses a DO loop to fill the atomic scattering \( f \) and electron propagator \( \beta \) matrices for a given energy. It then calculates the Green’s function, \( \gamma \), where \( \gamma \) is the identity matrix [4]. By default, \( \gamma \) is inverted via LU factorization, and \( \beta \) is found via LU solving (with back-substitution) using the LAPACK routines cgetrf and cgetrs [3]. Since DO loops are a GPU niche and CULA allows LAPACK to be called on a device, fms is a good candidate for GPU conversion.

2.2. Tiling: A CUDA algorithm
To fill 
 the device rather than the host, the symmetric \( mxn \) matrix storing a cluster’s weighted inter-atomic distances must exist in device global memory; the easiest way to accomplish this is to create 
 the device. Each element, where \( k \) is the complex wavenumber. First, a cluster’s list is downloaded into device global memory. Then each assigned to a thread, which will need 48B of global memory (2 s with 3 floats each). Reading each 2n times, once for every element in its corresponding row and column, would overtax the global memory bus. Using shared memory and tiling can mitigate the redundancy and congestion [1].

On a GPU, matrices can be divided into a regular lattice and assembled from a number of small, identical tiles. Each tile is a thread block with threads (with for warp efficiency) [2]. The largest square tile is a square warp (32x32). When tiling with square warps, each block needs 64 s; 32 for the rows it spans, and 32 for its columns. Each block can reduce its calls to global memory 32-fold by caching these 192 floats in shared memory, which is internal to its SM, and is quickly accessed by the entire thread block (though inter-block sharing is prohibited by rule 2) [1]. The caching of each float is handled by a thread. As only 192 threads per block (6 warps) need to participate, the caching is delegated to the first 6 warps via an if statement. Such inline control usually leads to GPU inefficiency, but extra SM passes occur only when threads within the same warp diverge [2]. After the cache is
filled, each thread calculates its . Finally, each warp performs a coalesced write to global memory (adjacent threads filling adjacent elements in ). Each column of tiles is a 1D grid (see fig. 1), with its own kernel call, because (rule 6) grids – a kernel’s work order – must have a regular shape.

**Figure 1.** Tiling with lattice locations , where . GPUs have two classes of redundancy. Small redundancies (checkered) are a necessary evil, since they usually require inline control to avoid. Cluster size is usually not a multiple of tile size, so tiled blocks often spill the boundaries of to fully cover it. Writing a control statement to avoid out-of-bounds memory access creates a divergence in at least n warps, so it is actually faster to allocate enough global memory to handle OOB indices and fill this extra space with garbage [2]. In contrast, large redundancies (striped), like filling both halves of a symmetric matrix, can be avoided (even though the additional small redundancy of sub-diagonal filling cannot). The ratio of the small redundancies to target data (solid) decreases substantially as grows much larger than tile width.

2.3. $Gfms$ (GPU/ fms)
The CUDA C kernels for filling and were incomplete at the time of publishing; the matrices were filled by the host into “pinned” host RAM, via modified FEFF 8.4 CPU code, then were copied into device global memory. Copying is fastest to/from “pinned” host RAM, which cannot be sent to a page file [1]. A simple CUDA C kernel calculated , and was created using CUDA to replace the original LAPACK calls. The GPU used was an Nvidia Tesla C2050 with error correcting code.

3. $fms$ (CPU) vs. $Gfms$ (GPU)
The runtimes of $fms$ and $Gfms$ were evaluated on crystalline MnO with varying cluster size . The matrix is , where ( is the largest orbital mapped). The device’s internal clock was used for device benchmarks and the function clock in “time.h” was used for CPU/host benchmarks.

Comparing table 1 with figure 2 shows the acceleration (a smaller exponent) that each section of $fms$ attained on the device, especially the time-limiting LU factorization (b). The amount of time the host spent controlling the device (fig. 2d) reveals that CUDA calls are not asynchronous, using many host clock cycles. Calling $Gfms$ asynchronously would require the host to use a CPU worker-thread for CUDA control; otherwise the host will get relegated to CUDA control until each CUDA call returns.

| Table 1. Runtime of Single-threaded CPU fms in milliseconds |
|---|---|---|---|
| CPU (ms) | Filling & (a) | cgetrf (b) | cgetrs (c) |

**Figure 2.** $Gfms$ runtimes on an Nvidia Tesla C2050. Several FEFF 8.4 processes used the same device simultaneously. When two processes attempted simultaneous use of the device, one was forced to wait, creating inconsistent timings (large error bars). A device queue should alleviate this problem. The early fit lines (dashed) are very fast, but the asymptotic fit lines (solid) are the true device complexity.

3.1 Anticipated Gains with On-device and Filling
In fig. 2, (a) and cgetrs (c) accelerated from to . It is reasonable to assume that the unfinished and filling kernel will also boost from to , yet retain its CPU coefficient, thus allowing the runtime of a completed Gfms to be estimated.

The SCF module can call Gfms hundreds of times. The host spends a lot of time preparing each energy level (k) for its Gfms call, much longer than Gfms runtime. If Gfms is called asynchronously, then as long as the device uploads its result before the host calls Gfms on the next energy level, Gfms runtime will be completely hidden by host prep. To prevent device downtime, multiple host threads could prep energy levels in parallel on multiple CPU cores. With x host threads, as long as the device time needed by each Gfms call is as long as the Gfms-prep time, there should be no conflicts in device usage. Both host and device will be constantly busy, and Gfms will realize the full GPU benefit.

The FMS module calls Gfms continuously (no prep). Gfms leaves nothing for the host to do but device control. Retaining some of the and filling on the host might balance the workload better.

![Figure 3](image)

**Figure 3.** Estimates of Gfms (a) and SCF (b, c) performance. A C2050’s 3GB of global memory sets a hard limit on N. In (d): actual use by Gfms (solid) and theoretical gains from an efficiency upgrade (dotted).

### 3.2 Error Considerations

Floating point arithmetic is non-associative [1]. Since blocks execute in random order, unit-in-the-last-place errors occur in GPU calculations, when compared to CPU results. For and , eq. 1 was used to compare a matrix uploaded from the device to one generated by the original FEFF 8.4 code (fig. 3 b,c).

(1)

![Figure 4](image)

**Figure 4.** fills quickly on the device (a). Direct CPU runtime was unavailable, as filled faster than the resolution (10 ms) of the CPU timing function clock. The anticipated CPU complexity is .

### 4. Discussion

FEFF 8.4 achieved significant performance gains on the GPU; a completed, asynchronous Gfms should expand them. Many XAFS algorithms are candidates for GPU acceleration (e.g. FEFF 9). Even cluster computers stand to benefit from GPU computing; a distributed host can control many devices.

### 5. References

[1] Sanders J and Kandrot E 2011 *CUDA by Example* Addison-Wesley
[2] Kirk DB and Wen-mei WH 2010 *Programming Massively Parallel Processors* Elsevier
[3] Ankudinov A L, Ravel B and Rehr JJ et al 2005 FEFF 8.4 Source Code
[4] Ravel B 2005 *Journal of Alloys and Compounds* **401** 1-2 pp 118-126