Block Conjugate Gradient solver in OpenCL

Kert Tali
Institute of computer science
University of Tartu
Tartu, Estonia
kert.tali@ut.ee

Eero Vainikko
Institute of computer science
University of Tartu
Tartu, Estonia
eero.vainikko@ut.ee

***Abstract*—The conjugate gradient method for solving certain systems of linear equations is widely used due to its iterative nature and fast convergence. Its boiled down algorithm contains simple matrix and vector operations which can be done in parallel with potential for great speedup. With the advent of GPGPU computing and accompanying programming models like OpenCL, basic linear algebra operations can be accelerated to beyond what was previously achievable on consumer grade hardware. This report presents and evaluates an OpenCL implementation of the block conjugate gradient algorithm with support for complex valued inputs and solving for multiple right-hand-side vectors in one go. The given code can be incorporated to any system that has an OpenCL supported device. The new implementation performs better than the platform adaptive clSPARSE library and implements features not found in other OpenCL alternatives.**

I. Introduction

A. The Conjugate Gradient method

Given a symmetric or hermitian positive-definite sparse matrix A and a right-hand side vector b, the system of linear equations in the form $Ax = b$ can be solved by using the Conjugate Gradient (CG) method. The method finds the solution to a convex optimization problem of the quadratic form

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{1}$$

in an iterative manner starting from some x [16]. The heart of CG is its usage of Krylov subspaces, which allow the algorithm to choose subsequent search vectors in A-orthogonal subspaces, from which it chooses the best vector in each dimension, finding the answer in n-iterations [16]. In practice, floating point round-off errors complicate that matter by introducing unwanted noise and delaying the convergence [16]. The rate of convergence is characterized by the condition number of the matrix or the difference between its largest and smallest eigenvalue [16]. A bad condition number can be helped by applying a preconditioner matrix, using the method called the Preconditioned Conjugate Gradient [16], but is not considered for this task. Main uses for such sparse linear equation solvers come from the discretizations of partial differential equations [15].

The derivation process of CG is rather complex and relies on several related fundamental methods like steepest descent and conjugate directions as well as thorough knowledge of linear algebra [16]. The final simplified formulation is shown in Algorithm 1 and comprises simple linear algebra routines. Namely the lines 1 and 5 need matrix vector multiplication. 3, 6, and 10 require vector dot product. 7, 8, and 12 show scalar vector multiplication and vector addition, more specifically the axpy routine.

Algorithm 1: Conjugate Gradient [16]

```
1  $r \leftarrow b - Ax$
2  $d \leftarrow r$
3  $\delta \leftarrow r^T r$
4  while not converged do
5      $q \leftarrow Ad$
6      $\alpha \leftarrow \frac{\delta}{d^T q}$
7      $x \leftarrow x + \alpha d$
8      $r \leftarrow r - \alpha q$
9      $\delta_{old} \leftarrow \delta$
10     $\delta \leftarrow r^T r$
11     $\beta \leftarrow \frac{\delta}{\delta_{old}}$
12     $d \leftarrow r + \beta d$
13 end while
14 return $x$;
```

A twist on this method is to adapt it to solve for multiple right-hand side (RHS) vectors. Mainly the algorithm stays intact, but we are solving for $AX = B$ instead, where X and B are $n \times b$ matrices. The benefits to this instead of solving for separate b vectors are a reduction in overhead and gains in speed due to denser calculations and less overall iterations and synchronization [13]. This is also known as the Block Conjugate Gradient algorithm.

B. OpenCL

Open Computing Language is an open standard and parallel programming model targeting CPUs, GPUs, FPGAs and DSPs [8]. Building parallel programs using OpenCL enables platform agnostic code, but one of the key points is its support for General Purpose Graphics Processing Unit (GPGPU) programming [9]. It may be beneficial for heterogeneous CPU+GPU platforms to delegate certain work off of the main processor, while a GPU's manycore architecture is excellent for parallelizing matrix and vector operations [9].

The lifecycle of an OpenCL program begins with the selection of a supported device or multiple devices on the

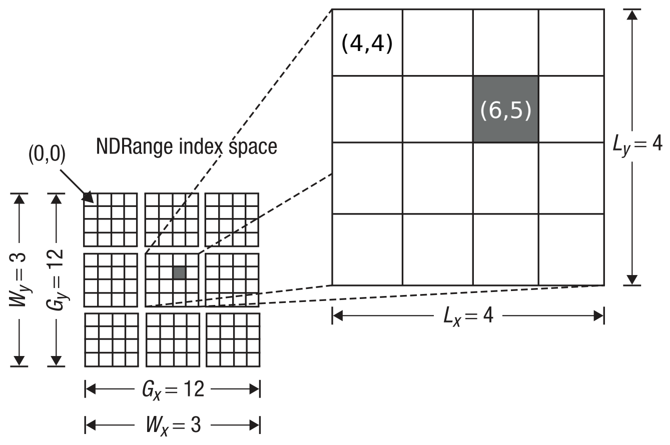


Fig. 1. A visualization of a 2D grid of 9 *work-groups* comprising 16 *work-items* each, each with their own 2D identifier. [11]

host machine. From there, the host program has to initialize a command queue, compile the device kernels, populate memory buffers, before orchestrating the execution.

Parallelism is achieved by the concurrent execution of kernel instances over the device's available compute units. The set of jobs running on a single compute unit and consequently execute strictly in parallel, share a local address space, and support synchronization is called a *work-group* [8]. Figure 1 is a visual representation of a 2D work grid (NDRange), with each work-item attributed a unique ID in the global index space – while useful for certain data-parallel algorithms, the range is usually single dimensional [8]. The device's implementation has set allowances on the maximum amount of *work-items* in a *work-group* based on the physical limitations of the compute units. Without communication between the groups and a limited amount of items running in parallel, designing kernels for tasks larger than a single *work-group* may prove to be difficult.

OpenCL as a standard does not guarantee the optimal performance of a program across all supported system configurations, leaving it to the programmer to understand the underlying platform and decide whether to create fast hardware-specific code or slow general code [11]. A good example are the hardware SIMD vectors (also known as *wavefront* in OpenCL or *warp* in CUDA) in GPUs, which execute a certain amount of work-items in low-level lockstep fashion, and can offer performance benefits if the kernel code takes them into account [11]. The API also exposes the efficient GPU texture memory with Image buffers, and inside *waves* programs can gain in memory read speed when coalescing the data read in a single clock cycle [12]. In addition to that, the implementations across platforms vary, implement older versions of the specification, or support a subset of operations, for example whether to support double precision floats. In conclusion the optimization of OpenCL programs is not trivial – automatic parameter choice and adaptive optimization for OpenCL being an active research area [17], [10].

In the five following sections we discuss details and considerations of implementing a CG solver in the OpenCL programming model. Section II looks at the linear algebra building-block operations required for each step. In Section III the most notable current implementations are compared and analyzed. Starting from Section IV implementation and novelty of the new solver is presented and it's tested in Section V.

II. Sparse and dense BLAS in OpenCL

Previously when giving a brief overview of CG and the condensed Algorithm 1, the main components were brought out. These were a subset of Basic Linear Algebra Subprograms (BLAS) as commonly known in the scientific computing community. Being essential to the implementation, they are discussed one by one.

A. *axpy*

The axpy routine represents the equation $y = \alpha x + y$, where x and y are vectors of length n and α is scalar. Since scalar-vector multiplication and vector-vector addition works in an element-wise fashion, this process can be executed with tiny granularity. In OpenCL each *work-item* would have a single dimensional index corresponding to its index in the vectors, resulting in the lines of kernel code shown in listing 1. `x` and `y` vectors reside in the device's global memory, which can be copied over from the host as arguments. Consequently, there are no problems running it over several *work-groups*, as each index is exclusive and all *work-items* can access global device memory. The routine does $n \times 2$ floating-point operations.

Listing 1. OpenCL kernel for calculating axpy

```
__kernel void axpy(__global float *x,
                   __global float *y,
                   const float a) {
    int idx = get_global_id(0);
    y[idx] += x[idx] * a;
}
```

B. *dot*

The dot product is the element-wise multiplication ($x^T y$) of two vectors of size n, with the values reduced into a single sum. A large and important part of this routine is the reduction, which is not trivial on OpenCL since *work-groups* cannot efficiently coordinate writing to some global memory address without causing a read-write race. A solution is to break the reduction into multiple steps, the first step being a local reduction within a *work-group*, creating an array with the size equal to the amount of *work-groups*. There are multiple ways of achieving local reduction, the simplest would be to let the first *work-item* of the *work-group* sum all the values into its place in a global output vector. The second step can then be executed even in the host if the vector is sufficiently small, or through another reduction kernel as needed.

C. SpMV

Sparse matrix-vector multiplication is the only second level BLAS operation needed, meaning it does not only deal with vectors and scalars, but is a combination vector and matrix operation. The operation is the equivalent of taking the dot product of each matrix row and the vector, resulting in a vector of size n. A sparse matrix is a matrix in which only a small portion of values are non-zero. Naturally this calls for distinct storage methods to avoid wasting unnecessary space on zeros. Computationally there are distinct differences in the algorithms used to handle these matrices as opposed to dense forms.

Although simple in concept, the optimization of this operation on GPGPU has been explored a lot [1], [2], [3], [6], [11] due to the large computational part it has in iterative numerical methods like CG. In practice, the operation is bottlenecked by its numerous and irregular memory accesses compared to flops [1]. The storage scheme choice has proven to be especially important when implementing SpMV in a massively parallel fashion, mainly as a clever way to speed up the data access using memory coalescing [3].

The Coordinate (COO) format is the most intuitive and general purpose storage method for sparse matrices, consisting of three arrays of size equal to the amount of non-zeros: values, row, and column indices [3]. Due to COO values not having a strict order, it has to be handled with the smallest granularity – one non-zero element per *work-item*. This introduces the problem of reduction of each row's sum into a single value. Since *work-groups* cannot communicate, usually a *segmented reduction* follow-up kernel is executed to do the aggregation on the device [3].

$$\begin{bmatrix} & 5 & 1 & \\ 2 & & 8 & 3 \\ & & 5 & \\ & 9 & 4 & \end{bmatrix} \quad \begin{array}{rl} \text{val} & \texttt{[2,5,1,8,9,5,3,4]} \\ \text{col} & \texttt{[0,1,2,2,1,2,3,2]} \\ \text{row} & \texttt{[1,0,0,1,3,2,1,3]} \end{array}$$

Fig. 2. Example storage of a sparse matrix in coordinate format.

Compressed Sparse Row (CSR) format is the most widely used representation, as it is the most compact while being general-purpose (*i.e.* not dependent on the distribution pattern of non-zeros) [3]. The structure is similar to COO in a sense that there is an array of non-zero values and an array of same length describing each value's column index, however the values are grouped and ordered by rows with a pointer array specifying the start index of each row's values. As opposed to COO, the rows are easily distributable to be handled by separate *work-items*, coarsening the granularity of parallelism, and eliminating the need to handle reductions. Greater flops are acquired if mapping several *work-items* to a single row with local reduction [3].

Other notable formats include [3]

- Diagonal – 2D data array, where the contiguous elements are the diagonals, with an offsets array indicating the offset of a row from the main diagonal. Fitting for use with specific diagonal matrices, as it suffers from great size unbalance when the matrix' shape is unfavorable due to padding used to fill any empty space.
- ELLPACK – Stores data as column-major dense matrix of non-zero row values along with an indices matrix showing the values' column index. Subject to imbalance as the values are padded to the largest amount of non-zeros in a single row of the sparse matrix, therefore it can lose in performance as some threads cease working early. It is efficient for vector architectures due to perfect coalescing and is handled by a *work-item* per row basis.
- Hybrid (ELL + COO) – This combination limits ELLPACK to a set amount of values and handling the few outlying elements using the COO format. This type of kernel generally gives the best flops in non-synthetic tests, but is difficult to set up.

$$\begin{bmatrix} & 5 & 1 & \\ 2 & & 8 & 3 \\ & & 5 & \\ & 9 & 4 & \end{bmatrix} \quad \begin{array}{rl} \text{val} & \texttt{[5,1,2,8,3,5,9,4]} \\ \text{col} & \texttt{[1,2,0,2,3,2,1,2]} \\ \text{ptr} & \texttt{[0,2,5,6,8]} \end{array}$$

Fig. 3. Example storage of a sparse matrix in compressed sparse row format.

III. Related works

The importance and performance benefits of multiple RHS iterative solvers shown in a more recent paper [5]. Authors stress the importance of multiple RHS vectors in electromagnetics, proposing a way of replacing the SpMV operation in the CG method with a Sparse Matrix-Matrix (SpMM) operation, where the RHS is a tall matrix instead. The result is a block based solver, which experiences less overhead than re-running the program for each RHS vector and shows much greater flops on manycore devices. Their implementation was based on CUDA and Intel MKL for benchmarking on NVIDIA GPUs and Intel CPUs respectively, the code of which was not made public.

Ready-made libraries for sparse BLAS enable using or implementing CG based solvers with relative ease. Some such solutions are clSPARSE on OpenCL, cuSPARSE and cusp for CUDA, and ViennaCL, a computing library running on multiple backends. Plenty of solutions and research has been published regarding CUDA, with much less available for the more general OpenCL platform.

A modest library for sparse matrix and vector operations (BLAS 1, 2, and 3) in OpenCL is clSPARSE [7]. It implements

- single- and double precision floating point values in kernels,
- SpMV with a CSR matrix,
- CG solver on real valued input with optional preconditioner.

However there is no support for complex values in its kernels.

A more general library containing OpenCL, CUDA and OpenMP implementations of sparse matrix operations and macros for Krylov solvers is ViennaCL [14]. The library implements

- single- and double precision floating point values in kernels,
- CSR, COO, ELL, HYB sparse matrices,
- Fast and adaptive SpMV,
- CG solver on real valued input with optional preconditioner,
- A wide range of preconditioners to choose from.

Support for complex values is a planned feature, though the project's maintenance has slowed down over recent years.

Similar form of heterogeneous matrix algebra library implemented on a variety of platforms is MAGMA [4]. However while it implements all kinds of complex and real valued kernels with even a multiple RHS $AX = B$ solver, the OpenCL implementation is missing sparse matrix operations specifically.

IV. Implementation

A solver was implemented in C and OpenCL, with the source code available for use[1]. In contrast to any other available OpenCL CG solver, it includes complex valued input and multiple RHS or block solving.

The code is split into device kernel code and host code which orchestrates device memory management and kernel executions. Each of the operations and the main loop in algorithm 1 is laid out in the host code as kernel execution calls to the device, which are essentially specialized BLAS routines. Similar approach can be seen in aforementioned libraries and in the CUDA cuSPARSE CG example. In contrast, the kernels could be designed having coarser granularity, but it makes sense to divide the responsibility into smaller portions to utilize *work-groups* more economically, promote reuse of code, and accompany the many synchronization points in the algorithm. As transferring data between different devices' memories is the main cause of performance loss in heterogeneous computing [10], large matrices and vectors are only transferred to the device at the beginning, and the result vector(s) queried in the end.

A. Kernels

Created kernels are contained BLAS operations which are designed with data-parallelism in mind. Many-core processors operating in SIMD-like manner (*e.g.* GPUs) should therefore be best suited.

The sparse matrix and vector multiplication kernel is analogous to the "Vector CSR" method described by Bell & Garland [3]. A CSR scheme is often used in general sparse matrix vector multiplication due to its stable performance with different sparsity patterns. The work is distributed by rows, where a fixed size group of *work-items* handles one row. The mentioned paper also describes a "Scalar CSR" kernel, in which one *work-item* handles one matrix row, however it is slower in most cases. The group of *work-items* can also be thought of as a *wavefront* with lockstep execution and some coalesced memory accesses. The size of the group should be

[1]https://github.com/ktali/cg-opencl

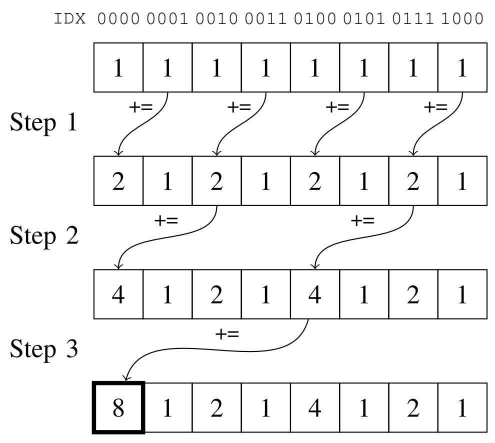


Fig. 4. In place reduction of an 8-element local array in parallel. This can be done in $\log n$ steps while utilizing the running threads.

chosen around the size of the average amount of non-zeros per matrix row, maximally the width of the SIMD lane of the device (if applicable), and be a value that is a power of two and a multiple of the *work-group* size. With the size of the *wavefront* fixed, there may be wasted cycles if the amount of non-zeros varies significantly over the rows as some threads may not have any values to process.

The vector-vector operations (dot product, subtraction, axpy and aypx) all operate with one vector index per *work-item*. With the exception of the dot product, all inputs and outputs are of the same dimensions and reside in global memory – trivial data-parallelism. The dot product is built on multiple steps of reduction.

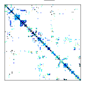
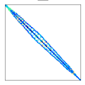
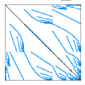
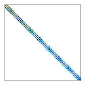
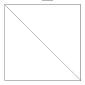
Dot product and the described sparse matrix-vector multiplication each require a form of sum reduction. For both cases, the local reduction is handled identically and in parallel like shown on figure 4 and listing 2. The matrix-vector row always fits in a *work-group*, therefore the given example is sufficient to calculate the result vector row-by-row. Same isn't true with the dot product, for which a vector could span multiple *work-groups*, requiring a second pass of reduction over a vector of size equal to the number of *work-groups*. In this implementation, the second reduction is calculated in the host code as the intermediate vector is significantly smaller in size and sending it to the host is of less concern.

Listing 2. Example of a parallel reduction in an OpenCL kernel.

```
int localIdx = get_local_id(0);
int mask = 1;
for (int offset = 1; offset < WG_SIZE;) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if ((localIdx & mask) == 0)
            localSums[localIdx] +=
                    localSums[localIdx + offset];
    offset <<= 1;
    mask <<= 1;
    mask += 1;
}
```

Although OpenCL has no native support for complex numbers, the usual course of action is to define the type

TABLE I
INPUT MATRIX PARAMETERS

	1138_bus	m_t1	parabolic_fem	mhd1280b	helm_fem
					
Type	real	real	real	complex (hermitian)	complex
Size (n)	1138	97 578	525 825	1280	16 384
Non-zeros (nnz)	4054	9 753 570	3 674 625	22 778	113 666
Row non-zeros (min–avg–max)	2–3.56–18	48–99.96–237	3–6.99–7	1–17.80–32	3–6.94–7

and operations yourself. Conveniently a complex number is represented with just two ordered floating point values, which can be encapsulated in a fixed size vector defined in the OpenCL specification as float2. The C host code can transfer its native complex type values in arrays as-is, as the pairs are held contiguously.

Block solving capability was built on top of the single RHS solver, now expecting a $n \times b$ dimensional matrix where b is the number of RHS vectors. It is expected that this dense matrix is tall and narrow. Kernels were modified to repeat calculations for all counterpart values in each of the vectors given. Some performance retaining steps were also taken. For example the SpMV kernel holds the static array of row sums in private memory registers of each *work-item* by defining the size b during compile-time.

V. Evaluation

The correctness of the implementation can be validated by substituting x in the system of linear equations. Matrices with smaller condition numbers are able to converge fine, which can also be seen by monitoring the iteration delta. With larger condition numbers, convergence may not occur at all, which could be helped by extending the implementation to a preconditioned conjugate gradient algorithm. The final result is also hindered by the maximum of 32-bits of floating point precision currently implemented, causing the accumulated error to throw off the accurate calculation of the residual and progress will stop after a certain amount of iterations. Fixes for these issues would be to extend the kernels to support double precision for less rounding errors, and implement the periodic residual reinitializing as shown in [16].

In the following section, the metric of performance will be presented in floating-point operations per second (flops). Flop counting is based on the matrix parameters, multiplied by the number of iterations and right-hand sides as shown in table II. Flops is acquired by dividing it by the observed wall-clock running time.

5 matrices with different size and sparsity were used for testing, an overview of which can be seen in table I. The specification of devices subject to testing is brought out in table III.

In figure 5, the performance comparison between the implementation and clSPARSE is shown. More specifically the library's single precision CG example code was repurposed for this experiment, which boasts device-adaptive kernel features. Missing from the results are the complex-valued matrices for clSPARSE due to missing support. The number of right-hand side vectors is fixed to 1 for the same reason. In both cases timing began after the device buffers had already been transferred. Wave size for GPU is chosen to be around the average number of non-zeros per row.

TABLE II
NUMBER OF FLOATING POINT OPERATIONS OF CG.

	operation	real	complex
once	SpMV	$2 \times nnz$	$8 \times nnz$
	sub	n	$2 \times n$
	dot	$2 \times n$	$8 \times n$
per iteration	SpMV ($\times 1$)	$2 \times nnz$	$8 \times nnz$
	dot ($\times 2$)	$2 \times n$	$8 \times n$
	axpy ($\times 3$)	$2 \times n$	$8 \times n$
	division ($\times 2$)	1	14

TABLE III
OPENCL DEVICES UNDER TEST

Device	Architecture	Cores	Memory	Clock speed
2080S	Nvidia "Turing"	3072	8GB	1.65–1.82 MHz
i5-8250U	Intel x64	4	16GB (System)	1.60–3.40 GHz

Larger inputs enjoy greater flops due to the large operational overhead of OpenCL even without the data transfer. For example, 1138_bus and mhd1280b have almost identical sizes, but one of them is much denser and has complex type, meaning much larger flops. Most significant improvement is seen on the GPU, with double the speed or more compared to clSPARSE. Even though the implementation was more focused on GPU performance, CPU does not fall short in any experiment. The SpMV *wavefront* size did not play a significant role in CPU speed, but was fixed at 1 since there is no SIMD execution that would benefit from it.

In figure 6, the effect of the block solver is shown in contrast to multiple runs of the algorithm on separate RHS vectors. The two matrices differ in terms of density and size. Performance increase of parabolic_fem dies off quickly probably due to there being more data transfer between the host and the device each iteration. This is due to the host-based reduction logic for dot product, requiring *work-groups* $\times\, b$ amount of elements to

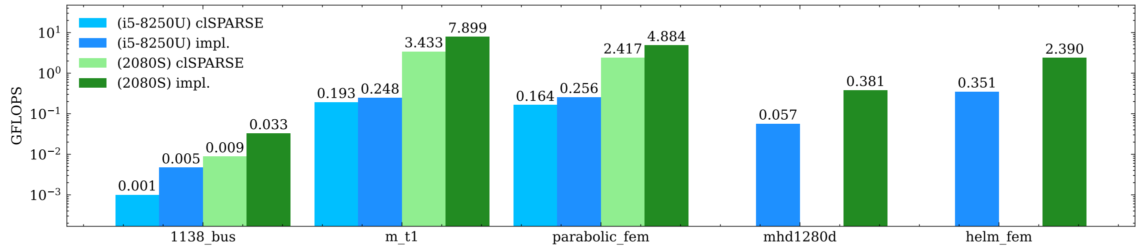


Fig. 5. GFLOPS comparison with clSPARSE's single precision CG running 5000 iterations (1 GFLOPS = 10^9 FLOPS)

be moved, where the amount of *work-groups* is defined by $\frac{n}{\textit{work-group size}}$. Matrix m_t1 is smaller while being denser and therefore the reduction transfer does not become a bottleneck as soon.

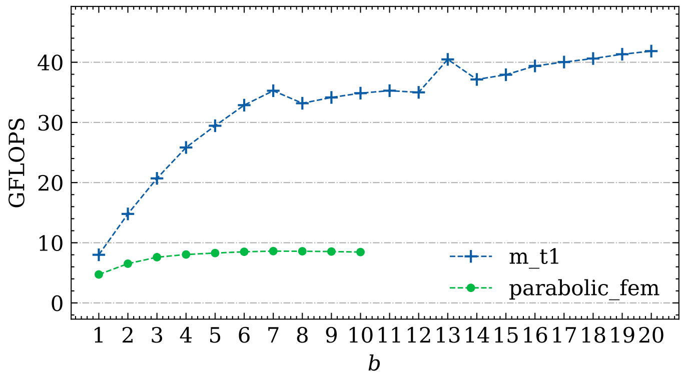


Fig. 6. Performance of the block solving on the 2080S with two matrices and increasing b.

The physical limits of this implementation are mainly related to the amount of right-hand sides, which may inflate some local and private arrays to sizes which exceed the device allowances. Testing showed that maximum achievable concurrent RHS on 2080S and matrix m_t1 was $b = 47$, with the performance by that point being 32.6 GFLOPS. Larger b prompted an `CL_OUT_OF_RESOURCES` error from the device.

VI. Conclusion

This work presented an ad-hoc implementation for a block CG solver, with effort put into optimizing it for GPGPU. Testing showed it exceeding the performance of the adaptive solver in the clSPARSE library with only a single RHS vector in both CPU and GPU test cases. The new solver also improved on the shortcomings of many OpenCL implementations with the introduction of complex-valued inputs.

As mentioned in previous works, the block-solving is a great way for improving performance. It achieved so plainly due to the density of data being fed to the device, with no special tricks in the kernel. Between running the program in single RHS batches and using a block solver, one can achieve upwards of four times the performance.

Some possible improvements include eliminating the need for host-device data transfer for the dot product reduction, as it hinders the performance on larger-dimensional matrices with more RHS vectors. This could be done by writing a separate reduction kernel, keeping more computation in the device. Implementing preconditioning would much improve the usefulness of this program as many inputs will not converge without it.

References

[1] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. *IBM Research Report RC24704*, (W0812–047), 2009.
[2] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.
[3] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.
[4] Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov. clmagma: High performance dense linear algebra with opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, pages 1–9, 2014.
[5] Adam Dziekonski and Michal Mrozowski. Block conjugate-gradient method with multilevel preconditioning and gpu acceleration for fem problems in electromagnetics. *IEEE Antennas and Wireless Propagation Letters*, 17(6):1039–1042, 2018.
[6] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):1–49, 2017.
[7] Joseph L Greathouse, Kent Knox, Jakub Poła, Kiran Varaganti, and Mayank Daga. clsparse: A vendor-optimized open-source sparse blas library. In *Proceedings of the 4th International Workshop on OpenCL*, pages 1–4, 2016.
[8] Khronos Group. *The OpenCL Specification*, 2020.
[9] Janusz Kowalik and Tadeusz Puźniakowski. *Using OpenCL: Programming Massively Parallel Computers*, volume 21. IOS Press, 2012.
[10] Frédéric Magoulès, Abal-Kassim Cheik Ahamed, and Roman Putanowicz. Auto-tuned krylov methods on cluster of graphics processing unit. *International Journal of Computer Mathematics*, 92(6):1222–1250, 2015.
[11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.
[12] C Nvidia. Opencl programming guide for the cuda architecture, version 4.2.
[13] Dianne P O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5(1-2):127–139, 1987.
[14] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jungel, and Siegfried Selberherr. Viennacl—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, 2016.
[15] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.
[16] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.
[17] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. Adaptive optimization for opencl programs on embedded heterogeneous systems. *ACM SIGPLAN Notices*, 52(5):11–20, 2017.