MultiGrid on FPGA Using Data Parallel C++

Christopher Siefert
Sandia National Laboratories
Albuquerque, NM, USA
csiefer@sandia.gov

Stephen L. Olivier
Sandia National Laboratories
Albuquerque, NM, USA
slolivi@sandia.gov

Gwendolyn Voskuilen
Sandia National Laboratories
Albuquerque, NM, USA
grvosku@sandia.gov

Jeffrey Young
Georgia Institute of Technology
Atlanta, GA, USA
jyoung9@gatech.edu

***Abstract*—Centered on modern C++ and the SYCL standard for heterogeneous programming, Data Parallel C++ (DPC++) and Intel's oneAPI software ecosystem aim to lower the barrier to entry for the use of accelerators like FPGAs in diverse applications. In this work, we consider the usage of FPGAs for scientific computing, in particular with a multigrid solver, MueLu. We report on early experiences implementing kernels of the solver in DPC++ for execution on Stratix 10 FPGAs, and we evaluate several algorithmic design and implementation choices. These choices not only impact performance, but also shed light on the capabilities and limitations of DPC++ and oneAPI.**

***Index Terms*—FPGA, OneAPI, Data Parallel C++, SYCL, Multigrid, Conjugate Gradient, Sparse Matrix Vector**

I. Introduction

Field Programmable Gate Array (FPGA) architectures have proliferated in domains such as embedded computing, cryptography, and multimedia processing. They allow for energy-efficient, application-specific acceleration and rapid prototyping of novel algorithmic approaches. However, their adoption in scientific high performance computing (HPC) has been limited, in part due to a lack of high-level programming language support. Many HPC application programmers are unfamiliar with hardware description languages (HDLs) like Verilog, and HPC programs are often too complex to practically code in such languages. OpenCL raises the level of abstraction somewhat, but its C interface still retains vestiges of hardware-level detail. Intel's oneAPI toolkit allows the use of the SYCL-based Data Parallel C++ (DPC++) on its FPGAs, CPUs, and GPUs. As DPC++ is based on a preferred HPC programming language, C++, oneAPI presents an excellent opportunity to revisit potential uses of FPGAs for HPC applications.

In this work, we consider the implementation of a multigrid solver, MueLu, in DPC++ targeting Intel FPGAs. We use a modified version of the high performance conjugate gradient (HPCG) benchmark [1] and its sparse linear algebra routines as a vehicle to test the solver with FPGAs. Section III explains our modifications to the HPCG code to better resemble real application use cases and to better suit the capabilities of the DPC++ programming model and supported architectures. The empirical results presented in Section IV compare the performance of algorithmic and implementation choices for key kernels in the code. Comparisons between CPU and FPGA currently favor the CPU, but we evaluate reasons for this performance gap and present strategies for narrowing this gap through further DPC++ optimizations.

II. Background

A. Data Parallel C++

The oneAPI programming model [2] includes the Data Parallel C++ (DPC++) dialect of the SYCL abstraction layer [3] for heterogeneous device programming. SYCL leverages generic programming capabilities of modern ISO C++ for cross-vendor code portability, and DPC++ offers some additional extensions particular to Intel devices. Kernels of work are expressed as C++ lambda expressions and submitted to a queue for execution on the device. Interfaces are provided for both explicit data management between host and device, and, where available, unified shared memory. Intel's LLVM-based oneAPI toolkit compiles a fat binary comprising both host machine code and device bitstream code, and the process requires no knowlege of low-level (i.e., HDL) device programming.

B. HPCG

HPCG [1], [4] is a benchmark code which executes a multigrid-preconditioned conjugate gradient (CG) method on a structured mesh. HPCG itself provides a reference implementation of the various kernels needed, while users are encouraged to supply optimized versions for their particular machine or architecture. Optimized HPCG benchmark performance results are reported biannually at the International Supercomputing Conference and the ACM/IEEE Supercomputing conference.

III. Approach

A. Modifications to HPCG

Some design decisions of the HPCG benchmark limit its ability to accurately represent certain classes of user applications. While HPCG uses data structures for matrix storage inspired by unstructured compressed sparse row storage, they are subtly different. Moreover, its multigrid method is explicitly structured with a very specific coarsening algorithm.

For this work, we modified HPCG to accept multigrid hierarchies generated by the MueLu [5], [6] algebraic multigrid library, which is part of the Trilinos [7] family of scientific software libraries. The multigrid algorithm is shown in Algorithm 1, where b represents the vector to which multigrid is applied, x represents output vector and A represents the matrix to which multigrid is applied. The restriction matrix, R moves data from the input "grid" to a coarse approximation space, while the prolongation matrix, P does the opposite. The Smoother function represents a iterative method such as

Gauss-Seidel. We also modified HPCG to more closely reflect real-world applications by enabling HPCG to:

- Use matrix-vector products for coarse grid transfers, rather than injection via array indexing.
- Use Chebyshev relaxation [8] as an alternative to the Gauss-Seidel smoother.
- Use a sparse direct solve via KLU [9].

Algorithm 1 x = Multigrid(A,b,level)

```
1: $x \leftarrow$Smoother$(A, 0, b)$ [Pre-Smooth]
2: $r \leftarrow b - Ax$ [SPMV, WAXPBY]
3: $r_e \leftarrow Rr$ [SPMV-T]
4: if level == max then
5:     $x_e \leftarrow A^{-1}r$ [LUsolve]
6: else
7:     $x_e \leftarrow$Multigrid$(RAP, r, level + 1)$ [Recurse]
8: end if
9: $x \leftarrow x + Px_e$ [SPMV, WAXPBY]
10: $x \leftarrow$Smoother$(A, x, b)$ [Post-Smooth]
```

Having modified HPCG, we implemented the algorithm in DPC++; For each vector and matrix operation within the CG solve proper (i.e., not the matrix load and setup), we implement a kernel that executes on the FPGA. Due to high overhead, the kernels avoid host/device memory transfers during the solve. The key, performance-relevant kernels are:

- WAXPBY - $w = \alpha x + \beta y$.
- SPMV - $y = Ax$.
- DotProduct - $\alpha = x^T y$.
- SGS - Symmetric Gauss-Seidel smoothing
- ChebyCore - Core operation of Chebyshev smoothing, $w = \alpha w + \beta d \odot (x - v)$.
- LUsolve - From KLU, solve $A^T x = b$.
- SPMV-T - $y = A^T x$.

Here A is a matrix, d, v, w, x and y represent vectors, α and β are scalars and $\odot$ indicates element-wise multiplication.

B. Kernel launch: parallel_for *versus* single_task

DPC++ offers two kernel programming models that map to the kernels listed in Section III-A: parallel_for and single_task. Quoting the DPC++ reference, the former executes "a kernel in parallel across a range of processing elements," while the latter executes "a single instance of the kernel with a single work item" [10]. While most of HPCG's kernels can be implemented in either form, the exceptions are (a) DotProduct, which is logically a parallel_reduce, but could also be implemented with single_task, and (b) SGS and (c) LUsolve, neither of which have a direct parallel_for implementation. The parallel_for construct is commonly used in performance portability libraries like Kokkos [11], while the oneAPI DPC++ FPGA Optimization Guide [12] stresses the use of single_task to maximize pipelining by the compiler. As we will show in Section IV-A, the parallel_for model generates faster code in all kernels we can implement with a parallel_for. Compilation reports indicate that the single_task kernels were pipelined, but we have not yet attempted manual optimizations.

C. Using stored transpose to avoid reordered writes

Sparse matrix-vector multiplication (SPMV), $y = Ax$, forms a key component of the multigrid method. It is used to transfer matrices between different levels of the hierarchies and can also be used in some classes of smoothers, such as Chebyshev relaxation. For a matrix stored in compressed-sparse row (CSR) format, the matrix is read sequentially, the output vector, y, is written sequentially, and the input vector, x, is effectively random access. On CPUs, read caching of the x vector a key performance strategy. FPGAs can also utilize read caching in this way.

For the restriction matrix, R, multigrid codes either explicitly store R and multiply with it directly, or implicitly store R, by storing P and using a transpose-mode SPMV (SPMV-T). In the implicit case, the matrix reads remain sequential, but now the input vector, x, is sequential while the output vector, y, is effectively random access. CPUs then utilize write caching of the output vector, y, to achieve good performance. Unfortunately, oneAPI on FPGAs does not support write-caching—only loads can be cached. As we show in Section IV-B, the performance penalty is substantial. Thus, we advocate using the explicit storage of R to improve performance and, as MueLu does, extend HPCG to support this option.

D. Reducing resource usage for non-critical code sections

One of the strengths of FPGAs is that prefetching, coalescing and caching behavior can be controlled at the level of individual load-store units (LSUs). These features can be selectively enabled to improve performance or disabled to decrease resource usage via template parameters to the ext::intel::lsu class. In our case, we focus on decreasing resource usage in the LUsolve kernel. In the Laplace2D problem we consider in Section IV, LUsolve represents 0.4% of the overall multigrid application time (and even less of the overall application time). We can thus economize resources in the LUsolve kernel with minimal changes to the overall runtime. To that end, we disable load caching for all operations in the LUsolve kernels by explicitly requesting loads to be done via uncached loads, ie ext::intel::lsu<ext::intel::cache<0>>::load(). The effect of this choice on performance and resource utilization is discussed in more detail in Section IV-C.

IV. Results

We consider two test problems derived from Trilinos/MueLu [5]. The first, Laplace2D, represents the discretization of a 5 point 2D Laplacian stencil on a 100×100 mesh, with a total of 10,000 unknowns. The second, Brick3D, represents the discretization of a 27 point 3D Laplacian stencil on a $40 \times 40 \times 40$ mesh, with a total of 64,000 unknowns. Unless stated otherwise, we use CG preconditioned with a three-level multigrid method using a single sweep of Chebyshev smoothing (pre and post) and a direct solve on the coarse level using KLU [9]. With the exception of some of the results in Section IV-B, we use the explicit transpose where we explicitly store the R matrix. Each problem is run one hundred times on the target machine. Within a run, each kernel is executed roughly 500 times (depending on CG's convergence behavior).

Each kernel execution is individually timed and a queue wait is executed (for oneAPI) before the end time is recorded.

Our CPU tests run on an 18-core Intel Xeon W-2295 (Cascade Lake) CPU with a 3.0GHz base frequency. Our FPGA tests run on a machine with two 16-core Intel Xeon Gold 6226R (Cascade Lake) CPUs, each with a base frequency of 2.9 GHz, and an Intel Stratix 10 SX FPGA. The CPU tests are compiled with the 2022.0 release of the OneAPI toolkit while the FPGA tests are compiled with the 2021.4 release with the FPGA toolkit for S10 programmable accelerator cards (PAC).

The maximum reported frequency of our HPCG implementation is 350 MHz, as reported by OneAPI and Quartus compilation reports. Based on empty kernel tests, we estimate the cost of a oneAPI kernel launch to be 7.9 μs on the CPU and 22.0μs on our target FPGA while empty queue wait costs are 2.0 μs on the CPU and 6.0 μs on the FPGA. By contrast, CPU function call overhead is measured in nanoseconds.

A. *Kernel launch:* parallel_for *vs.* single_task

We first consider the Laplace2D problem using two different implementations of our kernels, one using single_task for all kernels and the second using parallel_for for all kernels except DotProduct and LUsolve (Section III-B). As the results are similar across kernels, we show only the Chebyshev smoother implemented with parallel_for and single_task in Table I. For the Chebyshev smoother, parallel_for is 1.9x faster than single_task.

Method	Mean time	Std. Dev.
Chebyshev parallel_for	7.52e-4	9.3e-6
Chebyshev single_task	1.43e-3	1.1e-5

TABLE I
TIME PER CALL FOR CHEBYSHEV SMOOTHER USING PARALLEL_FOR AND SINGLE_TASK IMPLEMENTATIONS ON THE FPGA FOR LAPLACE2D.

Second, we compare the performance of symmetric Gauss-Seidel (SGS) and the Chebyshev smoothers on Laplace2D. Since SGS cannot be implemented in a parallel_for framework, we expect performance to be worse than Chebyshev, which is shown clearly in Table II. Chebyshev times are different for the pre-smoothing and post-smoothing operations, due to an optimization in the pre-smoothing that avoids an initial matrix-vector product.

Smoother	Mean time	Std. Dev.
Chebyshev-Pre	2.18e-4	1.93e-6
Chebyshev-Post	7.52e-4	9.29e-6
Symmetric Gauss-Seidel	6.51e-2	1.78e-5

TABLE II
TIME PER CALL FOR CHEBYSHEV AND SYMMETRIC GAUSS-SEIDEL KERNELS ON THE FPGA FOR LAPLACE2D.

B. *Using stored transpose to avoid reordered writes*

We consider the Laplace2D problem using two versions of the SPMV-T kernel: 1) an "implicit" version where we call a transpose-mode sparse matrix-vector multiplication routine with the matrix, $P = R^T$, and 2) an "explicit" version where we do a regular SPMV operation with the R matrix. Table III shows the mean time and standard deviation for the Laplace2D problem. As we can see, the explicit transpose is 28.8x faster, due in part to the sequential (as opposed to random) nature of its writes.

Smoother	Mean time	Std. Dev.
Explicit	5.17e-4	3.35e-5
Implicit	1.49e-2	9.19e-6

TABLE III
TIME PER CALL FOR SPMV-T KERNEL IN BOTH EXPLICIT AND IMPLICIT MODE ON THE FPGA FOR LAPLACE2D.

C. *Reducing resource usage for non-critical code sections*

In addition to timing results following the methodology described above, we also generated FPGA compilation reports using oneAPI 2021.4 to explore resource utilization of the LUsolve kernels for the direct solve at the bottom of the multigrid hierarchy using our FPGA implementation of KLU [9]. We use the oneAPI FPGA early image analysis tools to get utilization estimates [12]. As these do not perform full place and route, these tools enable fast design evaluation. Table IV compares a default implementation with one that explicitly disables caches on all LSUs. Within the kernel, we see a 39.6%, 40.3%, and 76.5% decrease in utilization of adaptive lookup tables (ALUT), registers (REG), and memory (RAM), respectively. Digital signal processor block (DSP) usage is unchanged and memory logical array blocks (MLAB) usage increases by a modest 17.6%. These decreases, when compared to the resource utilization of all other kernels combined, are significant, especially the memory usage, where the *RAM usage of the total application decreases by 43.2%*.

	Resource Utilization				
Method	ALUT	REG	MLAB	RAM	DSP
Default LUsolve	99,959.5	179,195	607	2,691	54
Uncached LUsolve	60,404.5	106,851	714	632	54
All Other Kernels	216,602	409,132	958	3,525	126

TABLE IV
ESTIMATED RESOURCE UTILIZATION FOR DEFAULT LSU IMPLEMENTATION AND UNCACHED LSUS FOR LUSOLVE.

We then compare the actual runtimes of the code with the default and uncached LUsolve using the Laplace2D problem on the FPGA in Table V. While the mean time is slightly lower in the default version, the difference is not statistically significant. Given the resource savings discussed above, we decided that this very minor performance degradation was acceptable.

Method	Mean time	Std. Dev.
Default LUsolve	4.25e-4	1.1e-5
Uncached LUsolve	4.35e-4	8.1e-6

TABLE V
TIME PER CALL FOR LUSOLVE KERNEL USING DEFAULT AND UNCACHED IMPLEMENTATIONS ON THE FPGA FOR LAPLACE2D.

D. CPU vs. FPGA Comparison

We now compare CPU and FPGA execution time. For the CPU execution we use four different configurations, two using DPC++ (with 1 core and 18 cores, notated oneAPI-1 and oneAPI-18), a serial version (Serial) that does not use oneAPI and a raw OpenMP version using 18 cores (OpenMP-18). We first consider the Laplace2D problem. Table VI shows the total solve time for each approach. The FPGA is roughly 4.1x slower than oneAPI-18 and 5.4x slower than oneAPI-1. The Serial and OpenMP runs are 7.4x and 21.5x faster than those done on comparable core counts using oneAPI. Launch/wait latency in μs (oneAPI) rather than ns (serial) plays a part here.

For Brick3D, the FPGA solve is significantly slower than oneAPI-18 and oneAPI-1, at 199.6x and 70.5x, respectively. However, the serial and OpenMP-18 runs are just 2.0x and 6.2x faster than their oneAPI counterparts. As Brick3D is 6.4x larger than Laplace2D, we would expect this problem to be less latency bound, more bandwidth bound, and benefit more from the use of threads. We note that the FPGA design runs at a frequency that is 8.6x slower than the tested CPU platform.

	Laplace2D		Brick3D	
Method	Mean time	Std. Dev.	Mean time	Std. Dev.
oneAPI-1	1.43e-3	3.47e-5	8.50e-3	3.87e-5
oneAPI-18	1.91e-3	1.18e-4	3.01e-3	1.49e-4
Serial	1.92e-4	2.42e-6	4.22e-3	2.06e-5
OpenMP-18	8.88e-5	6.05e-6	4.85e-4	2.74e-5
FPGA	7.76e-3	3.20e-4	6.00e-1	1.93e-4

TABLE VI
RUN TIME FOR MULTIGRID-PRECONDITIONED CG SOLVE FOR LAPLACE2D AND BRICK3D ON CPU AND FPGA.

V. Related Work

Our work lies at the intersection of algorithms (sparse multigrid solvers), programming models (DPC++), and heterogeneous computing (Intel FPGAs and the oneAPI toolchain). Zeni et al. [13] demonstrate an HPCG implementation using high-level synthesis on Xilinx FPGAs. That implementation operates only on the matrices generated by the HPCG benchmark harness, with a fixed number of nonzeros per row. Ours accepts arbitrary matrices as inputs and allows a variable number of nonzeros per row. Greisen et al. [14] evaluate sparse solvers on FPGAs in the context of video processing applications. Tsai et al. [15] present a DPC++ implementation of sparse linear algebra kernels, but using Intel GPUs rather than FPGAs. The DPC++ benchmark suite by Bavarsad et al. [16] targets FPGAs and includes a diverse set of applications including cryptography, image recognition, and 3D rendering.

VI. Conclusions and Future Work

DPC++ and the oneAPI toolkit raise the level of abstraction for FPGA programming considerably compared to alternatives such as Verilog, other HDLs, and even OpenCL. Our work demonstrates its applicability to a multigrid solver, a common scientific computing workload. We have shown how performance is sensitive to algorithmic and implementation choices and that some key challenges remain. The work of optimizing the code is ongoing, and future work includes evaluation of multinode execution and comparison to hand-coded low level implementations. We plan to investigate the use of unified shared memory to determine the impact on performance and programmability compared to the buffer/accessor-based data management used thus far. The early experiences reported in this paper reflect a nascent oneAPI software ecosystem, and we look forward to improvements in the components of that ecosystem, including compilers, tools, and documentation.

Acknowledgment

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] J. Dongarra, M. A. Heroux, and P. Luszczek, “High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems,” *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016. [Online]. Available: https://doi.org/10.1177/1094342015593158

[2] Intel Corporation, “Intel oneAPI programming guide,” December 2021, version 2022.1. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html

[3] Khronos SYCL Working Group, “SYCL specification,” June 2020, version 2020 provisional. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf

[4] The HPCG Team, “The HPCG benchmark website,” 2022 (acccessed January 11, 2022). [Online]. Available: https://www.hpcg-benchmark.org

[5] L. Berger-Vergiat, C. A. Glusa, J. J. Hu, M. Mayr, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner, “MueLu users guide,” Sandia National Laboratories, Tech. Rep. SAND2019-0537, 2019.

[6] ——, “MueLu multigrid framework,” http://trilinos.org/packages/muelu, 2019.

[7] The Trilinos Project Team, “The Trilinos Project Website,” 2020 (acccessed May 22, 2020). [Online]. Available: https://trilinos.github.io

[8] M. Adams, M. Brezina, and J. Hu, “Parallel multigrid smoothing: Polynomial versus Gauss-Seidel,” *J. Comp. Phys.*, vol. 188, pp. 593–610, 2003.

[9] T. A. Davis and E. P. Natarajan, “Algorithm 907: KLU, a direct sparse solver for circuit simulation problems,” *ACM Transactions on Mathematical Software*, vol. 37, pp. 36:1 – 36:17, 2010.

[10] Intel Corporation, “oneAPI Documentation,” 2020.

[11] C. R. Trott, D. Lebrun-Grandi, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, “Kokkos 3: Programming model extensions for the exascale era,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[12] Intel Corporation, “Intel oneAPI DPC++ FPGA Optimization Guide,” December 2021, version 20222.1. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html

[13] A. Zeni, K. OBrien, M. Blott, and M. D. Santambrogio, “Optimized implementation of the HPCG benchmark on reconfigurable hardware,” in *European Conference on Parallel Processing*. Springer, 2021, pp. 616–630.

[14] P. Greisen, M. Runo, P. Guillet, S. Heinzle, A. Smolic, H. Kaeslin, and M. Gross, “Evaluation and FPGA implementation of sparse linear solvers for video processing applications,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 8, pp. 1402–1407, 2013.

[15] Y. M. Tsai, T. Cojean, and H. Anzt, “Porting a sparse linear algebra math library to Intel GPUs,” *CoRR*, vol. abs/2103.10116, 2021. [Online]. Available: https://arxiv.org/abs/2103.10116

[16] N. N. Bavarsad, H. M. Makrani, H. Sayadi, L. Landis, S. Rafatirad, and H. Homayoun, “HosNa: A DPC++ benchmark suite for heterogeneous architectures,” in *39th IEEE International Conference on Computer Design (ICCD 2021)*, 2021, pp. 509–516.