OpenCL Performance on the Intel Heterogeneous Architecture Research Platform

Steven Harris
Roger D. Chamberlain
Christopher Gill

Steven Harris, Roger D. Chamberlain, and Christopher Gill, “OpenCL Performance on the Intel Heterogeneous Architecture Research Platform,” in *Proc. of IEEE High-Performance Extreme Computing Conference* (HPEC), September 2020.

Dept. of Computer Science and Engineering
McKelvey School of Engineering
Washington University in St. Louis

OpenCL Performance on the Intel Heterogeneous Architecture Research Platform

Steven Harris
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, MO, USA
0000-0001-5943-3036

Roger D. Chamberlain
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, MO, USA
0000-0002-7207-6106

Christopher Gill
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, MO, USA
0000-0003-0366-8586

***Abstract*—The fundamental operation of matrix multiplication is ubiquitous across a myriad of disciplines. Yet, the identification of new optimizations for matrix multiplication remains relevant for emerging hardware architectures and heterogeneous systems. Frameworks such as OpenCL enable computation orchestration on existing systems, and its availability using the Intel High Level Synthesis compiler allows users to architect new designs for reconfigurable hardware using C/C++. Using the HARPv2 as a vehicle for exploration, we investigate the utility of several traditional matrix multiplication optimizations to better understand the performance portability of OpenCL and the implications for such optimizations on cache coherent heterogeneous architectures. Our results give targeted insights into the applicability of best practices that were designed for existing architectures when used on emerging heterogeneous systems.**

***Index Terms*—Design space search, HARP, High-level synthesis, SGEMM, Field-programmable gate array**

I. Introduction

Even with nearly exponential increases in computer performance, matrix multiplication remains a persistent challenge and the advent of new accelerators reopens questions of both expression and optimization. With the breakdown of Dennard scaling, along with thermal and memory barriers, there has been a divergence from the performance improvements previously driven by Moore's law, and CPU clock rates have reached a plateau around 4-5 GHz. Under these constraints, both users and manufacturers have elected to scale horizontally. However, vertical and horizontal scaling of CPU and GPU resources has not been enough to keep up with the demands of modern workloads.

This performance gap has motivated some users to explore alternative architectures and bypass CPU and GPU architectures entirely by adapting their workloads to Application-Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs), which tend to be more performant than general purpose CPU/GPU devices for specific applications. With a focus on dynamic workloads, the reconfigurability of FPGAs has many advantages over fixed ASIC architectures. Yet, even though FPGAs have considerably less cost and development overhead than an ASIC, developing FPGA-based implementations can involve a steep learning curve that has traditionally only been accessible to those with knowledge of Hardware Description Languages (HDL) and digital system design techniques. That learning curve may be exacerbated in heterogeneous environments due to additional challenges such as orchestration and data migration.

This work supported by NSF grants CNS-1763503 and CNS-1814739.

To alleviate these challenges, OpenCL allows rapid development of programs that can execute across a wide range of heterogeneous platforms including, but not limited to, CPUs, GPUs, and FPGAs. The OpenCL framework enables computation orchestration on existing systems and its compatibility with the Intel High Level Synthesis compiler allows users to architect new designs for reconfigurable hardware using C/C++.

As the world transitions toward application-specific accelerators for modern workloads, Intel has developed an alternative accelerator in the form of a unified hybrid CPU+FPGA. One realization of this effort is the Heterogeneous Architecture Research Platform (HARP), version 2 of which consists of an Intel Broadwell Xeon CPU combined with an Intel Arria 10 GX1150 FPGA into a Multi-Chip Package (MCP) that enables shared DRAM memory through a single Intel QuickPath Interconnect (QPI) and two Peripheral Component Interconnect Express (PCIe) channels. Using the HARPv2 as a vehicle for exploration, we investigate the design space of matrix multiplication, using several existing cache-oriented optimizations to better understand the performance portability of OpenCL and the implications for such optimizations on this and future heterogeneous architectures.

Across a range of matrix sizes, we show that several classic optimizations designed for traditional caches are also effective on the HARP system. This includes transposition, blocking, and loop unrolling. When all optimizations are included, our implementations consistently outperform the optimized standard library implementation (CBLAS). However, there is considerable variability in performance, across both matrix sizes and various tuning parameters, that are not yet well understood and warrant further investigation. Given that FPGAs have been shown consistently to achieve considerable speedups over traditional microprocessor solutions across a wide range of applications [1]–[8], our results show a pressing need for a deeper investigation into underlying architectures created by the automated HLS process.

II. Background and Related Work

Exponential data growth has given rise to powerful analytic tools that can interpret and extract actionable information from expanding data sets [9]. Challenging tasks ranging from computer vision and natural language processing, to self-driving cars and social network filtering, typically rely on Machine Learning (ML) algorithms such as Deep Convolutional Neural Networks (DNNs) [10]. However, these techniques incur large costs in time and computational resources [11]. Information extraction for Big Data workloads inherently requires linear algebra, which can be challenging to implement with both low latency and high resource utilization. Many linear algebra libraries provide building blocks for these challenging numeric computations. Among the oldest and best known are the Basic Linear Algebra Subroutines (BLAS) and the Linear Algebra PACKage (LAPACK) which serve as the basis for many ML algorithms and mathematics libraries [12].

A. FPGAs

Several decades ago, FPGAs saw little adoption for many high-precision linear algebra workloads given their limited ability to handle high-precision floating point operations [13]. Since that time, there have been significant improvements in high-precision floating point performance for FPGAs [14], [15]. Consequently, modern FPGAs, such as Intel's Arria 10, have adopted variable-precision DSP blocks that include hardened floating-point operators. These architectural innovations enable the FPGA to processes both high-precision fixed-point and single- and double-precision IEEE-754 compliant floating-point operations efficiently. Unlike CPUs and GPUs with fixed datapaths, pipelines, and computational units, FPGAs allow users to adapt the hardware to critical features of computation. Of specific relevance to our problem of interest, Zhuo and Prasanna [16] have deployed matrix multiplication on an FPGA using HDL, as have Thomas and Luk [17] in the context of random number generation. Rather than using low-level languages such as Verilog or VHDL, our focus is on exploring the utility of a higher-level language, OpenCL.

B. Intel Heterogeneous Architecture Research Platform

No single device excels at all computational tasks, and computations can alternate between serial and parallel execution leaving the performance improvements of accelerators diminished by data migration that limits computational performance (and may be further exacerbated by imperfect coordination of multiple devices). Ultimately, most of these devices are limited by the speed of the PCIe interface. One radical architecture, which may minimize data migration in the case of FPGAs, comes from a solution which combines CPU and FPGA architectures. Intel has introduced the Heterogeneous Architecture Research Platform version 2 (HARPv2) which consists of an Intel Broadwell Xeon CPU combined with an Intel Arria 10 GX1150 FPGA into a Multi-Chip Package (MCP) with shared DRAM memory through a low latency, high bandwidth, Intel QuickPath Interconnect (QPI) and two Peripheral Component Interconnect Express (PCIe) busses. This supports a common last-level cache and DDR memory. In addition to unified DDR memory, the FPGA supports cache coherence and virtual-to-physical memory address translation. This provides a unique communications capability between the CPU and FPGA. This device gives users an opportunity to architect their own solutions without having to perform the arduous task of designing new circuitry from schematics to fabrication. Some examples of applications deployed on the HARP system include Convolutional Neural Networks (CNNs) [18], high-throughput DNA sequencing [19], dynamic programming [20], coordinative sparse LU factorization, and speculative implementations of breadth first search, single-source shortest path, Kruskal's minimal spanning tree, and Delaunay mesh refinement [21].

C. Open Computing Language

The OpenCL framework offers high-level abstractions that remove requirements for low-level hardware configuration and enable orchestration of memory and execution models for parallel workloads across accelerators. Source code can be written, compiled, and executed on a range of OpenCL compatible devices. Every OpenCL program has three primary components: Compute Units, Kernels, and Data Buffers. OpenCL generalizes heterogeneous devices into an OpenCL Platform model. One *Host*, typically a CPU, controls multiple *Compute Devices*. These *Compute Devices* contain multiple *Compute Units* which have multiple *cores*. Each *core* is typically an execution unit referred to as a *Processing Element* and each *Processing Element* can be used by one work-item. Work-items can be arranged into workgroups using an abstraction called an NDRange. OpenCL programs, called *Kernels*, are executed on multiple *Processing Elements*. The host sends kernels to the compute units and associates data buffers with compute unit(s) memory hierarchies. In many instances, when the hardware allows, the number of kernels sent to the compute device can be proportional to the dimensions of the data to be processed.

OpenCL provides two abstractions for partitioning workloads: NDRange and Single-Work-Item (SWI). An *NDRange* describes a 1- to 3-dimensional space for work-items. Contrasting this is the *SWI*, which follows a sequential model similar to many programming languages and is based on dependency analysis at compile time, OpenCL can extract pipelined parallelism from code to replicate a deeply-pipelined workflow that is common for FPGAs. The overall execution model for both methods is characterized in Figure 1.

III. Methods

All experiments are conducted on the Intel HARPv2 system at the Texas Advanced Compute Center (TACC). The programs are coded in OpenCL, conforming to version 2.0 of the specification [22]. A number of commonly used practices for matrix multiplication on multicore devices are applied to dense matrices that range in size from 1024×1024 to 8192×8192. The common advice for FPGA programming recommends writing code in the SWI format, allowing the

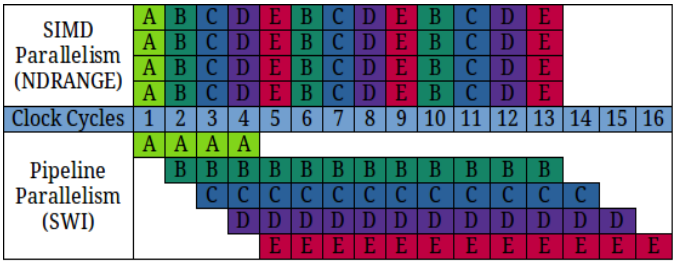

Fig. 1: Execution behavior.

compiler to identify elements that may be pipelined to take advantage of parallelism on the FPGA [23]. This is in contrast to the approach on GPUs which are naturally well suited to the NDRange methodology [24]. We explore both approaches. For NDRange implementations, the literature encourages users to set up a workgroup size that partitions the workload across processing elements in a uniform manner [24]. We utilize a method wherein the workgroup sizes are representative of the blocking and loop unrolling sizes. For instance, in kernel L2_B16_U16_ndrange, the kernel divides the matrices into 16×16 blocks and processes 16 elements simultaneously. This configuration will have a workgroup size of 16 by 16 processing elements. Each of the 256 processing elements takes a 16×16 block of the matrix and processes 16 elements each clock cycle.

To ensure correctness, each computation performed by the FPGA is compared against the same computation performed on the host processor using the `cblas_sgemm` function of the well-known CBLAS library. The various implementations investigated are organized into progressive levels as articulated below. At each level, both SWI and NDRange execution techniques are explored.

A. Level 0 – Naïve Implementation

The naïve implementation consists of the classic 3 nested loop implementation shown in Algorithm 1. Even though our performance expectations are low for this design, it forms a baseline for comparison with what follows. We will consider this the *unoptimized* version.

Algorithm 1 Naïve_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K)$

```
1:  C[M,N] = 0
2:  for row = 1 to M do
3:      for column = 1 to N do
4:          sum = 0
5:          for index = 1 to K do
6:              sum = sum + A[row,index] * B[index,column];
7:          end for
8:          C[row,column] = sum
9:      end for
10: end for
```

While a naïve implementation traditionally may not be classified as an optimization, it provides a good baseline to determine if future optimization choices are beneficial or detrimental to the performance of the computation. Referring to Algorithm 1 we see that the computation consists of three sequential for loops that range over the indices of the matrices.

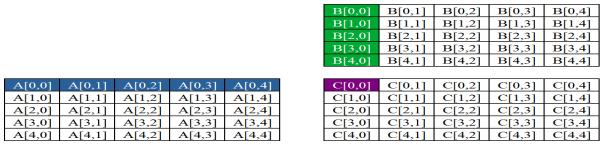

Fig. 2: Naïve Algorithm – matrix multiplication.

Looking at the matrix multiplication operations graphically, as shown in Figure 2, each element of the resulting matrix consists of one row of matrix A and one column of matrix B. This sequential method has a significant impact on the cache behavior. Disregarding cache line sizing and instead focusing on data arrangement within each cache line, we see that this algorithm makes poor use of the cache for columns of matrix B as shown in Figure 3.

Fig. 3: Naïve Algorithm – cache behavior.

Notice that in the case of Matrix A, all the elements required for the computation fit into an arbitrary length cache line. However, in the case of Matrix B, only one element of the B column is available in the cache line. This will cause significant cache misses and cache reloading which will negatively affect the performance of the overall computation. In the case of sequential matrix multiplication, notice that the row-major order allows us to retrieve elements of A in an efficient manner, but given that we require column entries of B we suffer from numerous cache misses equal to the dimension of the matrices themselves. To eliminate these misses, we will be performing a transposition of the B matrix which will streamline our element retrieval.

B. Level 1 – Transposition

The first optimization that we employ is to transpose the B source matrix. The data will be reordered on the host before it is sent to the device. In OpenCL (which is based on C/C++) matrices are stored in row-major order. As a result, when the B matrix is accessed down a column, there are significant inefficiencies in the cache usage. Algorithm 2 shows the resulting implementation, which benefits data locality for accesses to matrix B.

With the data arranged in row-major order, the computation now operates on contiguous row elements as seen in Figure 4 which benefits both spatial and temporal locality. With the transposition of matrix B we see efficient cache behavior as shown in Figure 5.

Algorithm 2 Transposition_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K)$

```
1:  C[M,N] = 0
2:  for row = 1 to M do
3:      for  column = 1 to N do
4:          sum = 0
5:          for index = 1 to K do
6:              sum = sum + A[row,index] * B[row,index];
7:          end for
8:          C[row,column] = sum
9:      end for
10: end for
```

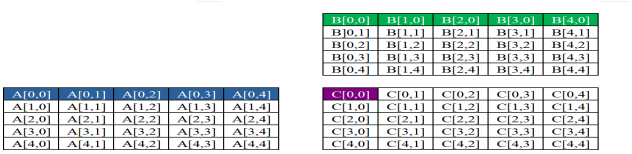

Fig. 4: Transposition Algorithm – matrix multiplication.

Fig. 5: Transposition Algorithm – cache behavior.

One interesting caveat about transposition is that it is not a cure-all for cache misses. Even with matrices being loaded into the cache in row-major order, we can still suffer from cache misses when the rows of the matrices are larger than the length of the cache lines. This is a hardware limitation that can significantly affect the performance of our computation.

C. Level 2 – Blocking

The blocking of data is a method that is beneficial to the computation irrespective of whether the data undergoes transposition or not. The key idea is to split the dataset into smaller partitions to be worked on independently. This is shown in Algorithm 3. Blocking benefits both temporal and spatial locality. We implement blocking with $TILE\_SIZE \in \{2, 4, 8, 16, 32, 64\}$.

As shown in Figure 6, when we perform blocking on the standard naïve matrix, we are taking advantage of data reuse. However, our data reads for matrix B still suffer from inefficient data access as we are only using one element out of an arbitrary cache read.

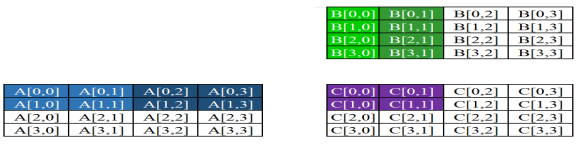

Fig. 6: Blocking Algorithm – naïve matrix multiplication.

Algorithm 3 Blocking_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K, TILE\_SIZE)$

```
1:  C[M,N] = Asub[TILE_SIZE] = Bsub[TILE_SIZE] = 0
2:  tile1 = tile2 = TILE_SIZE
3:  for  k2 = 0 to N by tile2 do
4:   for j2 = 0 to N by tile2 do
5:    for i2 = 0 to N by tile2 do
6:     for k1 = k2 to k2 + tile2 by tile1 do
7:      for j1 = j2 to j2 + tile2 by tile1 do
8:       for i1 = i2 to i2 + tile2 by tile1 do
9:        for i = i1 to i1 + tile1 do
10:        for j = j1 to j1 + tile1 do
11:          index = 0
12:          for k = k1 to k1 + tile1 do
13:            Asub[index] = A[i * K + k]
14:            Bsub[index] = B[j * K + k]
15:            index++
16:          end for
17:          for k = k1 to k1 + tile1 do
18:            C[i * N + j] += Asub[index] * Bsub[index]
19:            index−−
20:          end for
21:        end for
22:       end for
23:      end for
24:     end for
25:    end for
26:   end for
27:  end for
28: end for
```

After transposition, subdividing our data into blocks allows for more efficient cache usage given the spatial and temporal locality of the data for each read as shown in Figure 7.

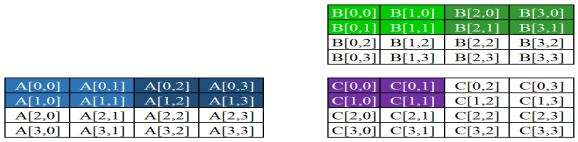

Fig. 7: Blocking Algorithm – transposed matrix multiplication.

D. Level 3 – Loop Unrolling

The final optimization we perform is loop unrolling. This is supported in the development toolchain via a `#pragma` statement. The unroll level is specified as one of 2, 4, 8, 16, 32, or 64. As with all of the other optimization levels, this is implemented in both the SWI and NDRange implementations. Loop unrolling allows us to increase the computational throughput. A standard computation is shown in Algorithm 4 and will take 1000 iterations to complete. An unrolled version of the standard computation is shown in Algorithm 5. By simply unrolling the computation once, we are able to complete the computation in 500 iterations. As the maximum unroll factor is

only limited by the architecture, this can allow for significant performance gains.

Algorithm 4
Traditional For Loop

```
1: for index = 0 to 1000 do
2:     purge(index);
3:     ...
4: end for
```

Algorithm 5
Unrolled For Loop

```
1: for index = 0 to 1000 by 2 do
2:     purge(index);
3:     purge(index + 1);
4: end for
```

Given the many optimization options and levels, so as to not suffer a combinatorial explosion of experimental configurations, we apply the above levels of optimization in a cumulative manner. As such, level 2 optimizations are all applied on code that has already been optimized at level 1. In addition, the loop unroll factor in level 3 is tied to the blocking factor used in level 2. So if we are performing a blocking size of 2, then we will unroll the computation by the same amount.

The full set of experiments is as follows (along with their labels):

1) Level 0 – naïve, both NDRange and SWI (indicated with L0_ndrange, L0_swi)
2) Level 1 – transpose matrix B, both NDRange and SWI (L1_ndrange, L1_swi)
3) Level 2 – blocking size in $\{2, 4, 8, 16, 32, 64\}$, both NDRange and SWI (L2_B2_ndrange, L2_B4_ndrange, L2_B8_ndrange, L2_B16_ndrange, L2_B32_ndrange, L2_B64_ndrange, L2_B2_swi, L2_B4_swi, L2_B8_swi, L2_B16_swi, L2_B32_swi, L2_B64_swi)
4) Level 3 – loop unrolling factor in $\{2, 4, 8, 16, 32, 64\}$, both NDRange and SWI (L3_B2_U2_ndrange, L3_B4_U4_ndrange, L3_B8_U8_ndrange, L3_B16_U16_ndrange, L3_B32_U32_ndrange, L3_B64_U64_ndrange, L3_B2_U2_swi, L3_B4_U4_swi, L3_B8_U8_swi, L3_B16_U16_swi, L3_B32_U32_swi, L3_B64_U64_swi)

The labels encode the relevant information to identify each experiment: the number after the L indicates the optimization level, the number after the B indicates the block size, and the number after the U indicates the unroll factor.

IV. Performance Results

To provide an appreciation of the breadth of performance results, Figure 8 plots the execution time for the matrix multiply operation as a function of matrix size (for square matrices) in every case we consider in this work (including the CBLAS result). At first glance it is clear that there is significant variability among the different kernels. To investigate this variability, we will separately address subsets of the kernels to help us elucidate and characterize this behavior, starting with the unoptimized naïve kernel.

An important thing to note in this plot is that the software-only CBLAS performance is in the highest performing group. This implies that a large number of the kernels do not provide performance that is competitive with well-tuned library code executed on traditional processor cores.

The level 0 (naïve) implementation performance results occupy the middle ground of the overall performance graph (kernels L0_ndrange and L0_swi). Not surprisingly, this unoptimized kernel does not provide performance that is competitive with other kernels.

We next turn our attention to the SWI kernels. All but one of this set are bunched in the upper left corner of the graph, indicating that they performed the worst of all those considered. The single exception is the L1_swi kernel. It is worth pointing out here that the SWI approach is the one most recommended for initial implementation by the manufacture's Best Practices Guide [23]. For the highly parallel task of dense matrix multiplication, this approach is clearly not the best one to pursue.

The best performing kernels are the NDRange kernels. The performance for many of the kernels bifurcates into two comparable groupings for the majority of matrix sizes. Figure 9 zooms in on the smaller matrix dimensions (6144×6144 and smaller) and includes only NDRange kernels.

Looking at these kernels, we see that the bifurcation starts almost immediately. All kernels in the upper diverging path, except for L3_B2_U2_ndrange, do not have the level 3 optimization (i.e., loop unrolling). Interestingly, L3_B2_U2_ndrange has the same performance profile as L3_B2_ndrange, and their execution times differ only by a fraction of a second. We assume that is because the L3_B2_U2_ndrange kernel has a loop unroll factor of only 2. We conclude that greater loop unrolling is critical for this application.

As we look at NDRange performance between sizes 3072×3072 and 6144×6144, we can see the general trend of bifurcation with the exception of the aforementioned L3_B2_U2_ndrange kernel. Notice that kernels without the loop unroll optimization continue along smooth gradations towards higher execution times but all kernels with the level 3 optimization have a spike in execution time at matrix dimensions of 4096×4096 and 6144×6144 while decreasing for the 5120×5120 dimension. This is an illustration of a pattern that happens frequently, in which we realize large swings in performance for unexpected reasons.

As we move forward to the larger matrix sizes, we see some interesting behavior starting after 7168×7168. Figure 10 shows the results zoomed in to these matrix sizes. For kernels with level 3 optimizations, those with a loop unroll factor of 2, 4, 8, and 16, have yet another spike which has a profile that degrades their performance even over kernels with only level 2 (i.e., blocking) optimizations. However, this behavior does not seem to impact level 3 kernels with unrolling factors of 32 or 64. Notice that for 8192×8192 matrices, the performance of

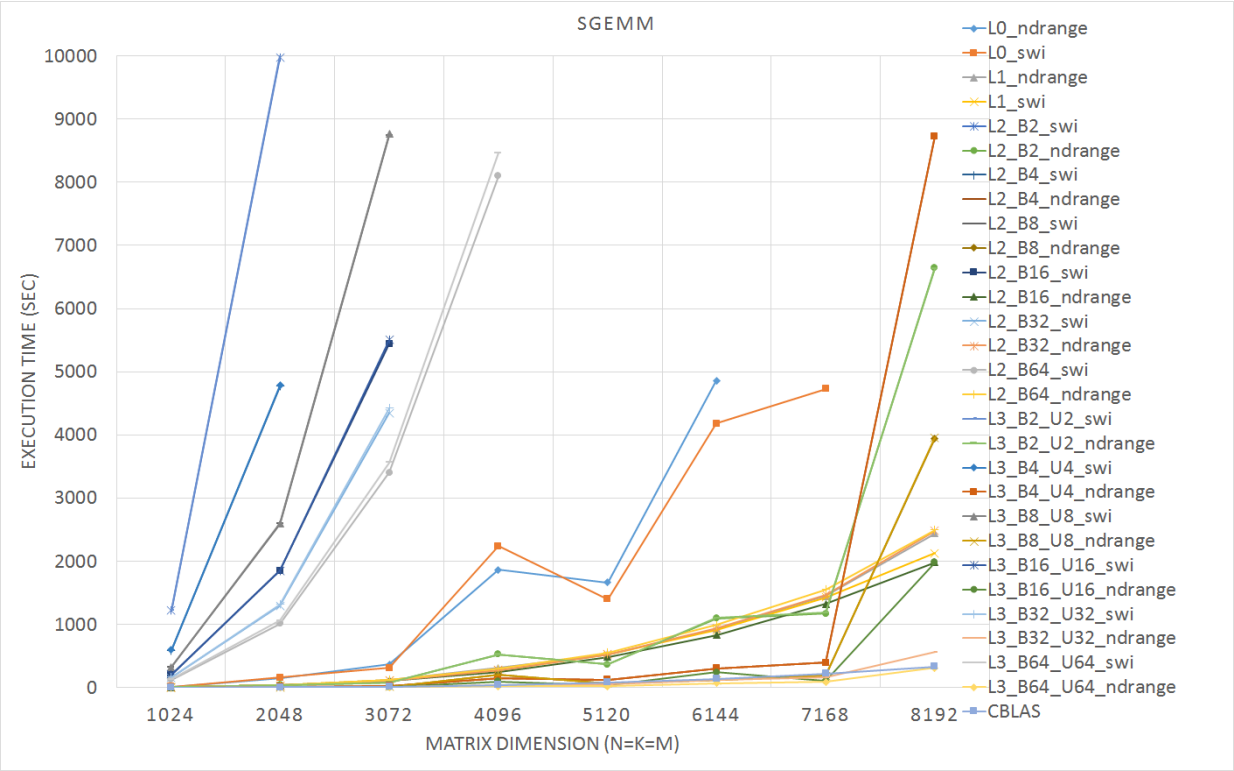

Fig. 8: Performance results – execution time vs. matrix size.

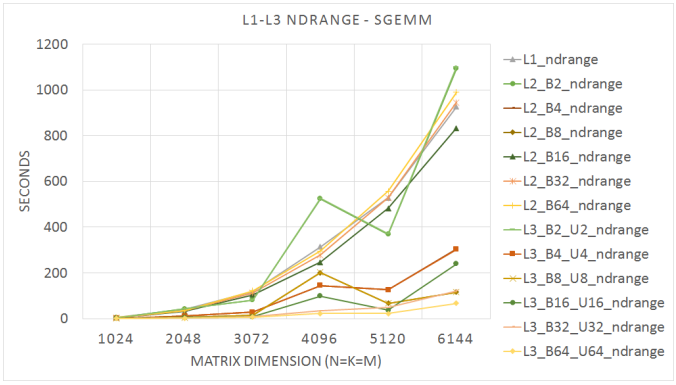

Fig. 9: Performance results for NDRange small matrices (6144×6144 and smaller).

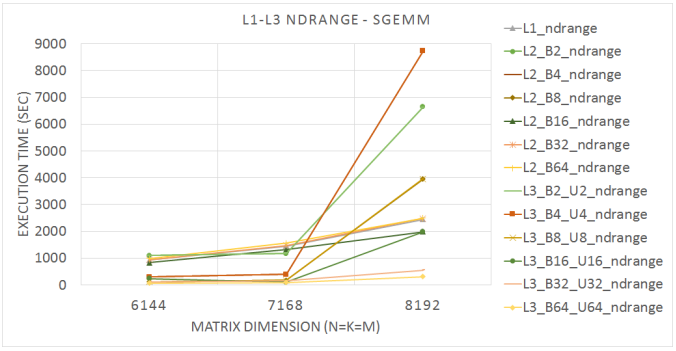

Fig. 10: Performance results for large matrices (larger than 6144×6144 inclusive).

L3_B16_U16_ndrange is comparable to L2_B16_ndrange and is a similar trend that we saw in L3_B2_U2_ndrange.

We next review the three highest performing kernels at each matrix size, comparing them to those conducted on the CPU. In all of our experiments, the highest performing kernel is L3_B64_U64_ndrange. However, we noticed that the runner-up kernels vary based on matrix dimensions as shown in Figures 11-18.

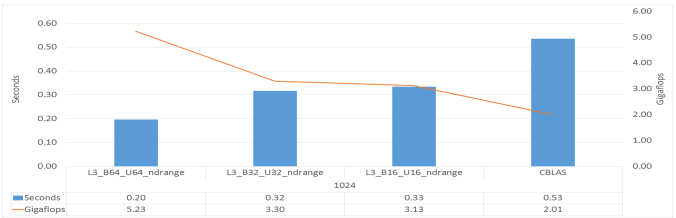

Fig. 11: Top 3 highest performance -– 1024×1024.

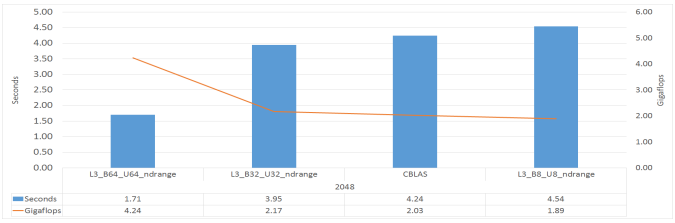

Fig. 12: Top 3 highest performance — 2048 × 2048.

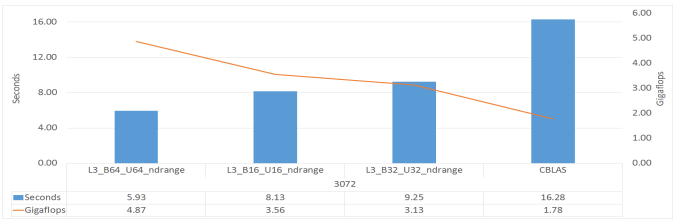

Fig. 13: Top 3 highest performance — 3072 × 3072.

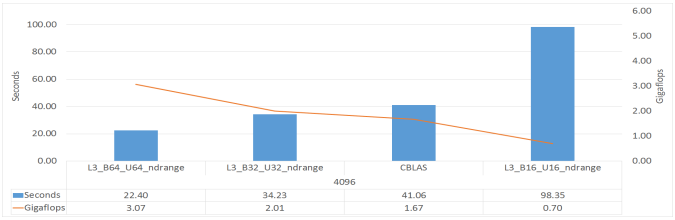

Fig. 14: Top 3 highest performance — 4096 × 4096.

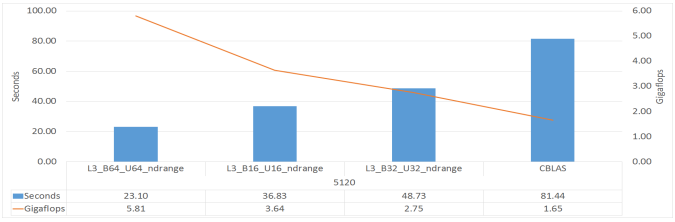

Fig. 15: Top 3 highest performance — 5120 × 5120.

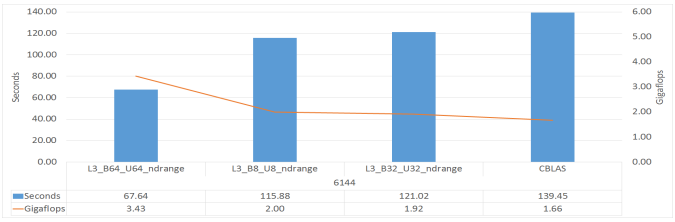

Fig. 16: Top 3 highest performance — 6144 × 6144.

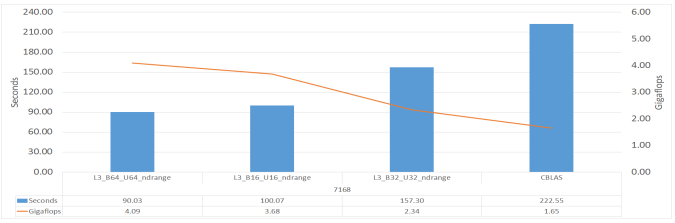

Fig. 17: Top 3 highest performance — 7168 × 7168.

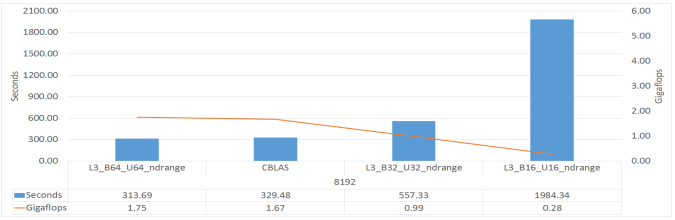

Fig. 18: Top 3 highest performance — 8192 × 8192.

While the particular runner-up kernel varies across matrix dimensions, what is consistent throughout is that the full suite of optimizations are needed for the FPGA deployment to be competitive with the CPU implementation. The top 3 in every case were level 3 optimizations that include both blocking and loop unrolling of an NDRange kernel.

As a final comparison, Figure 19 shows the execution time of the best-performing kernel, L3_B64_U64_ndrange, and the software CBLAS implementation.

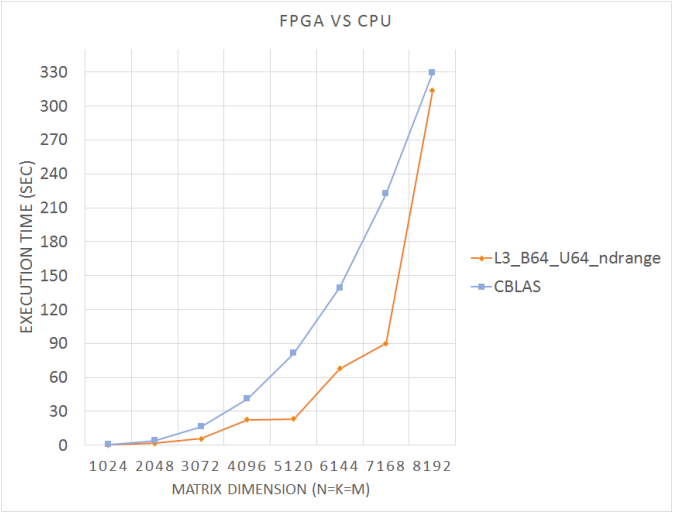

Fig. 19: FPGA vs. CPU execution time performance.

As one would expect, the execution time grows $O(N^3)$ with the dimension size N for both the software and hardware implementations. The software dependence upon matrix size, however, is fairly smooth, while there is considerable variability for the FPGA design. The FPGA design outperforms the software implementation at every matrix size, but the performance gain is highly variable, ranging from $1.05\times$ to $3.53\times$.

V. Discussion and Conclusions

The performance of the FPGA kernels varies considerably across optimization levels as well as matrix dimensions. This is in contrast with CBLAS, giving performance uniform and competitive across matrix dimensions. The results have shown conclusively that there are many considerations that must be taken into account in order to successfully develop high-performance kernels on reconfigurable hardware.

The SWI implementations, as seen in Figure 8, consistently performed worse than the standard CBLAS computations. The SWI execution model is recommended for FPGA implementations given that its architecture benefits pipelining, but after investigating Algorithm 3, we conclude that the compiler was unable to determine the exact loop iterations needed to pipeline the for-loop stages, due to our dynamic tiling, and executed many of the for-loops sequentially, leading to considerable serial execution stages.

The NDRange performance, as shown in Figures 9 and 10 has both interesting bifurcation patterns and varying performance spikes across dimensions. We speculate that this is caused by caches, memory subsystems, or underlying microarchitectural features, in effect hitting size boundaries of the various physical structures involved. A similar effect is frequently seen in GPU applications, where a mismatch between requested resources and available resources provided by the hardware can result in a zig-zag performance pattern as one parameter or another is varied [25], [26]. Since in our case we have a hardware data path that is constructed independent of the matrix size, and then utilized across a range of matrix sizes, it is reasonable to expect to experience this effect.

A general rule in optimization is to design your algorithms to make optimal usage of architectural features such as cache behavior and memory coalescing [12]. Given the dynamic design of our implementation using HLS it is difficult to determine how to ensure this *a priori*. We should not assume a particular cache size or method to coalesce memory reads. We would argue that our results clearly show performance sensitivity to this class of optimizations. Some block sizes actually degraded performance which we speculate was caused either by imbalanced memory access or inefficient cache usage. We would argue that HLS introduces the need for new design methods that may differ from our assumptions about traditional cache and memory hierarchies.

The kernel with the most aggressive optimizations, L3_B64_U64_ndrange, always had the best performance, but an interesting phenomenon occurred as we increased the dimensions of the matrices. Given that the L3_B64_U64_ndrange kernel, with the highest optimization, always finishes first, it may be natural to assume that the slightly less optimized kernels (L3_B32_U32_ndrange, L3_B16_U16_ndrange, and L3_B8_U8_ndrange) would occupy 2nd, 3rd, and 4th place respectively. For the 1024×1024 matrix, this is the behavior for the top 3 kernels. We encounter situations where the above does not hold, for instance, in the 5120×5120 matrix wherein the L3_B16_U16_ndrange kernel outperforms the L3_B32_U32_ndrange kernel. This becomes more pronounced for the 6144×6144 matrix where the L3_B8_U8_ndrange kernel outperformes the L3_B32_U32_ndrange kernel. When we reach the final matrix dimension of 8192×8192 only our L3_B64_U64_ndrange kernel performs better than the standard CBLAS computation.

We can conclude several things from these experiments:

1) In a system such as the HARP, in which the FPGA is tied in to the cache hierarchy, classic optimizations targeting cache behaviour are beneficial to the FPGA as well as the CPU.
2) In order to take advantage of the accelerator in this environment, all of the optimizations we consider are needed to achieve performance competitive with mature, optimized software.
3) The standard CBLAS library was able to outperform all but one optimized kernel in spite of the fact that these kernels are executing on an FPGA.
4) Many optimized kernels have degraded performance for a range of workloads. Whether or not a particular optimization ends up being performant is not at all clear prior to implementation and measurement.
5) These experiments confirm that FPGA performance can exceed CPUs such as the Intel Xeon class processor when coding in OpenCL, but achieving that performance benefit is not necessarily just a simple porting exercise.

We thus need to rethink whether general purpose tools give us sufficient flexibility to truly design, tailor, and reconfigure components of our particular computation. To make the most of accelerators, we must understand not only the algorithms but how they interact with data, workflows, and other cooperative components. Future work thus will need to include more effective compile-time performance models, so as to allow the tool chain to effectively do some of the investigated optimizations automatically (e.g., loop unrolling is quite common in compilers for CPUs, without asking the programmer to specify the degree to which loops are unrolled).

Acknowledgment

The authors thank both Intel and the Texas Advanced Computing Center (TACC) for access to the experimental hardware through the Hardware Accelerator Research Program.

References

[1] Rodriguez-Borbon, Jose, et al. "Field Programmable Gate Arrays for Enhancing the Speed and Energy Efficiency of Quantum Dynamics Simulations," J. Chem. Theory Comput., 16(4):2085–2098, 2020.

[2] Dhar, S., L. Singhal, M. Iyer and D. Pan. "FPGA Accelerated FPGA Placement," Proc. of 29th International Conference on Field Programmable Logic and Applications, 2019, pp. 404-410.

[3] Jovanović, Ž., and V. Milutinović. "FPGA accelerator for floating-point matrix multiplication," IET Computers & Digital Techniques, 6(4):249-256, July 2012.

[4] Stitt, G., R. Lysecky and F. Vahid. "Dynamic Hardware/Software Partitioning: A First Approach," Design Automation Conference, June 2003.

[5] Villarreal, J., D. Suresh, G. Stitt, F. Vahid and W. Najjar. "Improving Software Performance with Configurable Logic," Journal on Design Automation of Embedded Systems, 7(4):325-339, November 2002.

[6] Brebner, G. "Single-Chip Gigabit Mixed-Version IP Router on Virtex-II Pro," Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines, September 2002.

[7] Cardells-Tormo, F., et al. "Efficient FPGA-based QPSK Demodulation Loops: Application to the DVB Standard," Proc. of 12th International Conference on Field Programmable Logic and Applications, Sept. 2002.

[8] Hauser, J., J. Wawrzynek. "Garp: a MIPS processor with a reconfigurable coprocessor," IEEE Symposium on Field-Programmable Custom Computing Machines, April 1997, pp. 12-21.

[9] Reed, Daniel A., and Jack Dongarra. "Exascale computing and big data," Communications of the ACM, 58(7):56-68, June 2015.

[10] Rani, Kumari Seema, et al. "Deep Learning with Big Data: An Emerging Trend," 19th International Conference on Computational Science and Its Applications, IEEE, 2019.

[11] Liu, Weibo, et al. "A survey of deep neural network architectures and their applications," Neurocomputing, 234:11-26, 2017.

[12] Dongarra, Jack J., et al. *Numerical Linear Algebra for High-Performance Computers*, SIAM, 1998.

[13] Shirazi, Nabeel, Al Walters, and Peter Athanas. "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," Proc. IEEE Symp. on FPGAs for Custom Computing Machines, 1995.

[14] Govindu, Gokul, et al. "Analysis of high-performance floating-point arithmetic on FPGAs," 18th International Parallel and Distributed Processing Symposium, IEEE, 2004.

[15] Underwood, Keith. "FPGAs vs. CPUs: Trends in peak floating-point performance," Proc. of ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, 2004.

[16] Govindu, Gokul, et al. "Analysis of high-performance floating-point arithmetic on FPGAs," 18th International Parallel and Distributed Processing Symposium, IEEE, 2004.

[17] Thomas, David B., and Wayne Luk. "Multivariate Gaussian random number generation targeting reconfigurable hardware," ACM Trans. on Reconfigurable Technology and Systems, 1(2):12:1–12:29, June 2008.

[18] Zeng, H., C. Zhang and V. Prasanna. "Fast generation of high throughput customized deep learning accelerators on FPGAs," Proc. of International Conference on ReConFigurable Computing and FPGAs, 2017.

[19] Choi, Young-kyu, et al. "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," Proc. of 53rd Annual Design Automation Conference, June 2016.

[20] Cabrera, Anthony M., and Roger D. Chamberlain. "Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2," in Proc. of 7th International Workshop on OpenCL, May 2019.

[21] Li, Zhaoshi, et al. "Aggressive pipelining of irregular applications on reconfigurable hardware," Proc. of ACM/IEEE 44th International Symposium on Computer Architecture, 2017.

[22] Howes, Lee, and Aaftab Munshi. "The OpenCL Specification, version 2.0." Khronos Group, 2015.

[23] Intel® FPGA SDK for OpenCL™ Pro Edition: Best Practices Guide. Intel, April 2020.

[24] Owens, John D., et al. "GPU computing." Proceedings of the IEEE, 96(5):879-899, 2008.

[25] Zhang, Yao, and John D. Owens. "A quantitative performance analysis model for GPU architectures," Proc. of Int'l Symp. on High Performance Computer Architecture, Feb. 2011, pp. 382–393.

[26] Ma, Lin, and Roger D. Chamberlain. "A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines," Proc. of 23rd IEEE International Conference on Application-specific Systems, Architectures and Processors, July 2012, pp. 24-31.