HAND-OPTIMIZED FPGA IMPLEMENTATIONS OF NPBENCH KERNELS

Carl Friess, Rahul Goli, Fabian Landwehr, Aditya Manglik, Mengtao Zhang
Mentor: Alexandros Nikolaos Ziogas

Department of Computer Science, ETH Zürich

ABSTRACT

Field Programmable Gate Arrays (FPGA) offer an alternate computation platform in addition to traditional CPU- and GPU-based systems. Unfortunately, it is difficult to fully realize the advantages of FPGAs due to programming challenges characterized by, for example, the lack of automatic memory management and programmer-transparent caches. To this end, various frameworks aim to compile code written in a high-level language such as Python to an FPGA-compatible bitstream. Evaluating these frameworks involves measuring how close to optimal the performance of an automatically generated implementation is, introducing the need for hand-optimized code as reference. This project provides such FPGA implementations of five numerical kernels taken from the NPBench [1] benchmark suite. We experimentally evaluate our implementations using synthesis, emulation and hardware execution on a state-of-the-art Xilinx FPGA. Our implementations achieve up to 10.7x higher performance than comparable CPU- and GPU-based versions.

1 Introduction

Python's high productivity has made it increasingly popular in scientific programming and high-performance applications. Nonetheless, Python applications fall short in performance compared to C, C++, or FORTRAN. NumPy [2] has emerged as the preferred library for accelerating numeric computations in Python. While NumPy predominantly uses optimized C implementations to execute array operations, a significant amount of the execution still takes place within the python interpreter, leaving room for significant performance improvement.

Various solutions have attempted to address this issue by generating compiled and optimized versions of Python programs, in particular through just-in-time compilation (e.g., Numba [3]), ahead-of-time compilation (e.g., Pythran [4]), compilation for CUDA GPUs (e.g., CuPy [5]) and multi-platform compilation based on dataflow execution representations (DaCe [6]). Given these diverse solutions for optimizing Python applications, it is challenging to compare language compatibility and performance among different frameworks and target architectures without a standard set of benchmarks.

NPBench [1] aims to address the lack of a standard for comparing performance among different frameworks by providing a suite of benchmarks consisting of numerical kernels from a wide range of areas. Each framework is compared to a respective NumPy implementation. Although this provides a valuable comparison, determining how close to optimal the performance of solutions generated by these frameworks are requires comparison with manually optimized versions. To solve this challenge, we provide hand-optimized implementations of five different NPBench kernels for a state-of-the-art Xilinx FPGA (Virtex UltraScale+ VU9P [7]).

Motivation. We consider FPGAs an important platform for high-performance applications [8,9]. Indeed, Microsoft's Catapult project [10], Intel's acquisition of Altera [11] and recently, FPGA availability on Amazon Web Services [12] are evidence that FPGAs are becoming an increasingly prevalent platform and show significant potential to improve the performance of some types of computations.

Related work. [13], and [14] are some of the latest Python compilers that aim to generate code optimized for execution on FPGAs. In this work, we implement hand-optimized versions of kernels that form the benchmarks for such frameworks.

2 Background

In this section, we introduce Xilinx's HLS toolchain and describe the methodology used to develop FPGA implementations of our kernels. We discuss each step in the development pipeline and state our approach to obtaining performance metrics. Finally, we briefly introduce each kernel that we include in this project's scope. The kernels Azimint, Durbin, Gram-Schmidt, Cavity Flow, and Conv2D were selected to cover a broad spectrum of problem domains considered in the NPBench suite.

High-Level Synthesis. We utilize the Xilinx HLS [15] framework that enables synthesizing FPGA-compatible netlists (also referred to as *bitstreams*) from high-level C++ code. Additional pragmas [16] provided by the Vitis HLS compiler allow control over the hardware layout that is synthesized. Since HLS code is C++ compliant, we test it for correctness by compiling it for CPU execution. This process is in some cases referred to as simulation but should not be confused

with the distinct process of hardware emulation.

We found the *hlslib* [17] library useful in streamlining various parts of the development effort and optimizing processing elements (PEs - sections of code that run in parallel to other PEs). *hlslib* provides wrappers for the instantiation of PEs, as well as FIFO queues (*Streams*) used to connect and synchronize individual PEs. The wrappers allow the simulation to mimic the parallelism of dataflow sections. *hlslib* provides vectorization support via the *DataPack* class, which can be instantiated with a parameter W specifying the width of the vector. DataPack allows us to operate on W elements in parallel. Furthermore, hlslib provides CMake files used to build the simulation binaries and run the initial synthesis.

Synthesis, Emulation, and Hardware. The complete HLS development workflow involves several steps. The *initial synthesis* converts HLS C++ code to hardware description files. It also calculates cycle counts that accurately reflect the *initiation interval* and *depth* for individual pipelines and overall latency for various sections of the code. Additionally, the toolchain applies heuristics to estimate propagation delay and maximum clock frequency without generating the full hardware layout. The runtime of the initial synthesis is reasonable, which was necessary given the limited time frame of this project. However, this process does not provide us with any feedback on the correctness of our implementation. We use the simulation described above to ensure correctness at this stage.

Next, we use *hardware emulation* to more accurately simulate the execution of our kernel on the target FPGA. This takes significantly longer to build and execute compared to the initial synthesis. Nonetheless, we have included these results in our project where feasible.

Finally, the kernels can undergo the full *hardware layout synthesis* allowing for actual hardware execution. Unfortunately, kernels that pass emulation may still fail to correctly execute due to complex timing constraints. Debugging the implementations is difficult and time-consuming due to insufficient instrumentation and the long runtime of full synthesis.

Performance metrics. Given the time constraints of this project, we primarily select the results from the initial synthesis to evaluate and compare our implementations. The cycle counts and the clock estimates are reasonably representative of the final values determined by full synthesis. Therefore, we consider these metrics sufficient to make meaningful comparisons between implementation versions and with other platforms. Nonetheless, we have used data from hardware emulation where feasible.

Next, we briefly introduce each kernel.

Azimint. Azimuthal integration is a mathematical operation used to perform data-reduction on area-detector frames of small- and wide-angle X-ray scattering experiments [18]. This task is time-consuming and sometimes limits the productivity of modern synchrotron beamlines. The input consisting of two arrays is summarised into a much shorter output array.

Durbin. Durbin or Levinson-Durbin recursion is an algorithm for finding the solution to an equation involving a Toeplitz matrix [19]. An array is passed as an input to the algorithm, which is used to create each row of the matrix and solve it iteratively.

Gram-Schmidt. The Gram-Schmidt orthogonalization algorithm is a fundamental linear algebra algorithm that performs QR Decomposition, which decomposes a given matrix into an orthogonal matrix and an upper-triangular matrix. It is often used for solving least square problems.

Cavity Flow. Cavity Flow is an algorithm for solving the Navier-Stokes equation for a two-dimensional cavity flow problem. The code takes the density and kinematic viscosity of the fluid as input and iteratively builds the momentum matrices u and v.

Conv2D. Discrete two-dimensional convolution is a mathematical operation that takes a matrix, and a kernel as inputs then composes them to obtain an output matrix. It is often used for signal processing or building deep neural networks which take images as inputs.

3 Methodology

We first describe our FPGA performance model, followed by the roofline analysis for our kernels. Next, we describe the optimizations considered for each kernel.

3.1 Performance Model

We employ the FPGA performance model described in [20]. Achieving peak performance on an FPGA involves maximizing utilization of the available resources on the reconfigurable fabric. The highest number of processing elements N that can be instantiated on a chip with available resources D_{max}, L_{max}, and B_{max} (denoting DSPs, LUTs and BRAM, respectively) is:

$$\begin{aligned} N_{max}(PE) = Maximize\ N\ subject\ to\ & N \cdot PE.D \leq D_{max} \\ & N \cdot PE.L \leq L_{max} \\ & N \cdot PE.B \leq B_{max} \end{aligned} \quad (1)$$

where PE.D, PE.L and PE.B are integer counts of DSPs, LUTs and BRAM in a single processing element.

Peak Performance: The peak number of operations per cycle for a pipelined program is the maximum clock frequency attainable by the implementation, multiplied by the maximum number of operations attainable among any processing element:

$$C_{FPGA,peak} = Maximize\ f \cdot Max_{PE}\{PE.A \cdot N_{max}(PE)\} \quad (2)$$

where f denotes the frequency. **Operational Intensity** for each kernel is calculated manually based on a static analysis of the implementation. **Memory Bandwidth** is assumed as the peak bandwidth available to the FPGA in the system. It is calculated by multiplying the bandwidth of a single DDR4 module (21.33 GBps), with the total number of DDR4 modules installed in the system (4).

Roofline. Roofline models [21] help characterize the performance improvements of a kernel's implementation on specific hardware. The roofline is defined as the effective throughput of computational elements with peak performance (F_{peak}) as a function of the number of operations per byte transferred to and from off-chip memory I, with bandwidth to external memory of R:

$$F(I) = min\{F_{peak}, R \cdot I\} \tag{3}$$

3.2 Azimint

Azimuthal integration iterates over its input arrays and produces a smaller output array summarizing the input data. The naive baseline implementation operates on a single output array element, repeatedly iterating over the entire input. The innermost loop that iterates over the input is pipelined but suffers from a high initiation interval of three cycles due to data dependencies.

Loop Reordering. Our first optimized version interchanges the nested loops such that the algorithm first iterates over the input, continuing the computation of all output elements for each input value, thus only loading the input arrays once. More importantly, reordering the loops resolves the data dependencies and allows all pipelines to operate with single-cycle initiation intervals.

Output Segmentation. Given that each output element is independent of the others, the next objective was thus to operate on multiple elements of the output in parallel. We obtain significant speedups due to increased parallelism by replicating the core computation in multiple PEs that operate on smaller output segments.

Loop Unrolling. The output array is generally of a manageable size, so we explored fully unrolling the innermost loop, allowing concurrent computation of all output array elements. Beyond further increasing parallelism, fully unrolling the innermost loop allows pipelining to be applied earlier, namely the outer loop that iterates over input elements. This step brings back the data dependencies initially resolved through loop reordering, resulting in a high initiation interval of nine cycles. We apply additional optimizations below to partially hide the increased initiation interval by increasing memory throughput.

Although implementing the unrolled computation in a single PE would be relatively simple, the runtime of the synthesis was no longer practical. We could achieve significantly lower synthesis times by segmenting the implementation into multiple PEs, as was done for output segmentation.

Vectorized Memory Accesses. Before the core computation can begin, the algorithm must find the maximum value in one of the input arrays. This simple pipelined operation accelerates over the entire input array with a single cycle initiation interval. In light of the previously mentioned optimizations, the runtime of this preliminary step becomes non-negligible. The memory interface of the FPGA is capable of loading 32 floating point values in each cycle. By using the `DataPack` type, we vectorize the memory accesses and compare 32 input elements in each cycle, thus significantly reducing the overhead of this initial search.

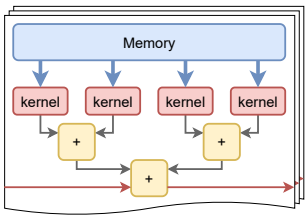

Fig. 1: Vectorization principle applied to the core computation of the azimuthal integration kernel.

The same principle is applied to the core computation. We load multiple values from memory simultaneously and perform the core computation for all loaded inputs in parallel. We then need to summarize these results before the next iteration of the pipeline can begin. We employ a network of operations visualized by figure 1 for a vector width of four. The resulting pipeline is characterized by a large depth but maintains the same initiation interval.

Towards Full Utilization. With the optimizations mentioned above, we have a parameterized design that is capable of exceeding full utilization of the FPGA. The challenge is thus to tune the parameters such that performance is maximized without exceeding the resource constraints. Despite reverting minor optimizations to reduce hardware utilization, we found four to be the highest possible vector width for the core computation. Due to extremely long synthesis times for some combinations of parameters, we could not fully explore the search space and insufficient reporting of the synthesis tool, which makes it challenging to develop heuristics. Hence, there may still be some room for improvement.

3.3 Durbin

The Durbin algorithm takes an array as input and iterates over its length, considering a sub-array of increasing size in each iteration. Each iteration computes a dot-product of the sub-array and reverses it. Every iteration depends on the output of the last hence, executing iterations in parallel is not possible. Therefore, we aim to make the individual iterations faster.

Dot-Product Optimization. The first optimization aims to reduce a data dependency introduced by the dot-product accumulation (*fadd*) operation. We create two separate PEs for computing the point-wise multiplication and the accumulation. These functions are connected using streams. The output of the second function is the output of the iteration.

Input Output Optimization. The second version of our Durbin algorithm treats each iteration of the code as a separate dataflow entity, which are joined by *streams*. The input is read from memory and pushed into a stream, and the output is pushed into another stream which will become the input of the next iteration. The dot-product is computed similarly as above without using any additional buffer storage.

However, as streams do not support LIFO operations and FPGAs do not operate using stack storage, the array reverse operation becomes a bottleneck. In order to reverse an array, we have to write the entire array to a buffer and then read the buffer in reverse (so this step is entirely sequential). If N is the size of the algorithm's input, then $7*N$ streams are used in this algorithm, making synthesis impossible for large values of N ($N > 100$).

3.4 Gram-Schmidt

The Gram-Schmidt algorithm requires a lot of read/write operations to memory. In each iteration, we need to update the values of the input matrix by subtracting the linear-dependent column from all remaining columns. Memory operations pose challenges for optimization on FPGAs since we can only read or write to a memory port once per cycle. Further, each column orthogonalization is dependent on the orthogonalization of all previous columns, leading to a lot of inter-iteration data dependencies. The original algorithm is shown in algorithm 1.

Algorithm 1: Original Gram-Schmidt algorithm

```
/* Matrix A is of size N*N      */
for i ← 1 to N do
    r_{i,i} = |a_i|;
    q_i = a_i / r_{i,i};
    for j ← i+1 to N do
        r_{i,j} =< q_i, a_j >;
        a_j = a_j − r_{i,j} * q_i;
```

Our baseline version implements the original algorithm. Even inside one single loop iteration, matrix A is dependent on the matrix R, and matrix R is dependent on matrix Q. Thus, the critical path in the computation DAG is extremely long. For the sake of optimization, we use another formulation of the Gram-Schmidt algorithm [22], which reduces the length of the critical path significantly (algorithm 2).

Algorithm 2: Improved Gram-Schmidt algorithm

```
/* Matrix A is of size N*N      */
for i ← 1 to N do
    r2_{i,i} =< a_i, a_j >;
    for j ← i+1 to N do
        rn_{i,j} =< q_i, a_j >;
    r_{i,i} = sqrt(r2_{i,i});
    q_i = a_i / r_{i,i};
    for j ← i+1 to N do
        a_j = a_j − (rn_{i,j} / r2_{i,i}) a_i;
    for j ← i+1 to N do
        r_{i,j} = rn_{i,j} / r_{i,i};
```

The improved algorithm does all the dot-product calculations, then simultaneously updates A, Q, and R. In our implementation, we reorder the loops with buffering to pipeline dot-product calculation with single-cycle initiation intervals. We further build a streaming framework around each iteration to activate loop-level concurrency and reduce overhead within each iteration. Nonetheless, the inter-iteration dependencies can not be solved due to the nature of the algorithm.

3.5 Cavity Flow

Cavity Flow can be described as a stencil algorithm. The algorithm has three main components: a) computing the Jacobian matrix (b), b) Poisson-Pressure Matrix (p), and c) momentum matrices (u and v). There is a circular dependence between the three steps: b requires a, c requires b and a requires c. These steps repeat for nt iterations. Algorithm 3 shows these three steps in action. In each of these steps, the value of the output matrix at $row = i$ and $col = j$ is dependent on the value(s) of the input matrix(ces) at positions $[i,j]$, $[i-1,j]$, $[i+1,j]$, $[i,j-1]$, and $[i,j+1]$.

Algorithm 3: Cavity Flow Naive Code

```
/* Matrices are of size NX*NY   */
b ← init_Jacobian(u_prev, v_prev);
p ← init_Poisson_Pressure(b);
for i ← 1 to NY − 1 do
    for j ← 1 to NX − 1 do
        u_{i,j} = comp_U_elem(u_prev, p, i, j);
        v_{i,j} = comp_V_elem(v_prev, p, i, j);
```

Buffering. This dependency is solved by adding extra buffers that reduce read dependencies on the p matrix. We use two separate buffers of p for computing u and v matrices. The first buffer stores the row above the one currently being computed (buf_1 in algorithm 4), and the second stores the row currently being computed (buf_2). This significantly reduces the dependence on memory and buffer accesses and speeds up the computation.

Algorithm 4: Cavity Flow Buffered Code

```
b ← init_Jacobian(u_prev, v_prev);
p ← init_Poisson_Pressure(b);
buf_1, buf_2 ← init_P_bufs(p);
for i ← 1 to NY − 1 do
    for j ← 1 to NX − 1 do
        u_{i,j} ← comp_U_elem(u_prev, p, buf_1, buf_2, i, j);
        v_{i,j} ← comp_V_elem(v_prev, p, buf_1, buf_2, i, j);
        buf_1, buf_2 ← update_P_bufs(p, i, j);
```

Vectorization. Furthermore, we tried to make improvements using vectorization by leveraging hlslib `DataPack` types. Each row is divided into vectors of size W. However, since values in the previous column and next column have to be accessed for computing each matrix entry, computing the 0^{th} and $W-1^{th}$ element of a vector requires accessing the previous and next vector in the same row, leading to reduced performance. Algorithm 5 shows the difference introduced by the vectorization.

Algorithm 5: Cavity Flow Vectorized Code

```
for i ← 1 to NY − 1 do
    for j ← 0 to NX/W do
        for k ← 0 to W do
            #pragma HLS UNROLL
            u_{i,j} ← comp_U_vector(u_prev, p, i, j, k);
            v_{i,j} ← comp_V_vector(v_prev, p, i, j, k);
```

3.6 Conv2D

The two-dimensional convolution convolves the input matrix and the kernel. The naive implementation multiplies each entry with weights from the kernel then sums them up in the output matrix. The intra-iteration data dependencies cause the pipeline to stall. Convolution is typical for systolic arrays, and there is much potential for parallelism. The optimization idea is as follows: reordering, buffering, scaling, and unrolling.

First, we reorder the loops and use a buffer to store the intermediate results in the innermost loop. This avoids pipeline stalling. Further, we scale up the innermost loop to prepare for the loop unrolling.

For unrolling the loops, we use extra buffers to store the weights and biases that we can reuse in the inner loops, then access the buffers instead of memory to resolve intra-iteration dependencies. In the end, we unroll the innermost loop by the scaling factor. Figure 2 shows the computation diagram after the optimization.

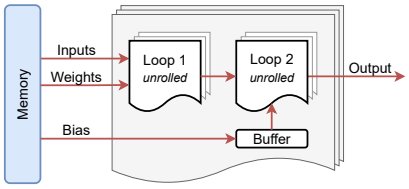

Fig. 2: Computation diagram of conv2d after optimization.

4 Experimental Results

In this section, we present the experimental evaluation of our implementations. We compare the overall runtime of each version of our kernels to the corresponding DaCe [6] implementations from NPBench [1]. Where available, we include the runtime of the GPU implementation.

We use our initial naive implementation of each kernel as our baseline. This version includes only minimal manual optimizations, such as initial pipelining of loops. We show one or two additional versions for each kernel, including the manual optimizations described in section 3.

Experimental setup. Our evaluation targets the Xilinx Virtex UltraScale+ VU9P FPGA in a machine with four individual 16GB DIMM RAM modules. This FPGA is capable of operating at up to 200MHz. We used the Vitis 2020.2 HLS compiler on Ubuntu 20.04.2 LTS for synthesis.

The CPU benchmarks were executed on a machine with two Intel Xeon Gold 6154 processors, each with 18 cores (36 cores total) clocked at 3GHz and 24.75MB of L3 cache. The GPU benchmarks were executed on an Nvidia V100 GPU.

Input Sizes. The NPBench framework specifies four different input sizes for each kernel: *S*, *M*, *L* and *Paper* [23]. We compare all four benchmark sizes but if not otherwise stated, we refer to the *Paper* benchmark, as it was used for the original NPBench paper [1].

4.1 Roofline Analysis

We illustrate the performance of each of our kernel implementations in the roofline plot in figure 3. We observe that our optimized kernels exhibit significant performance improvements with respect to the baseline implementation. For instance, successive optimizations of the Conv2D kernel improve the operational intensity and the peak performance on the target FPGA (both scales are logarithmic).

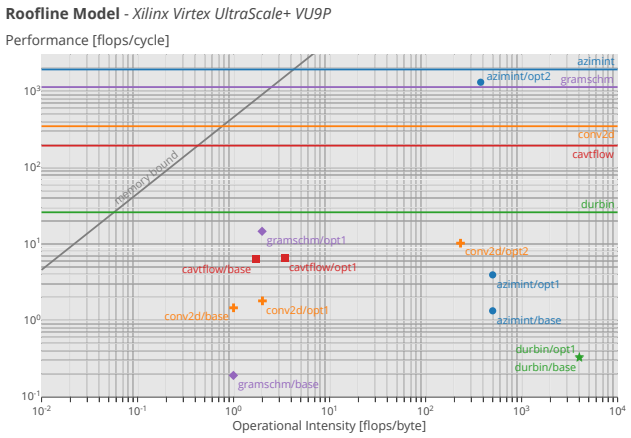

Fig. 3: Roofline plot showing each version of the selected kernels, as well as the peak performance for each kernel.

4.2 Azimint

Comparing our baseline to the first optimized version, where we applied loop reordering to reduce the initiation interval from three to one cycle, we expect to see a speedup of 3x. Indeed we achieve just over 3.5x speedup, with the excess relating to improved critical paths, lowering the clock period from 5.943ns to the target 5ns.

The second optimized version's extensive loop unrolling and vectorized memory accesses achieve 67% resource utilization (limited by LUTs). A large amount of parallelism leads to a speedup of **1564x** compared to the baseline. This version is **3.7x** faster than the CPU.

Azimuthal integration proved to be well suited for FPGA execution. Most operations can be scheduled independently, giving rise to much parallelism, which is needed to achieve high resource utilization and, thus, high performance.

4.3 Durbin

The optimized version in figure 5 uses streams to compute the dot-product. For small inputs, this version performs **5x** better than the baseline. However, for larger inputs, this

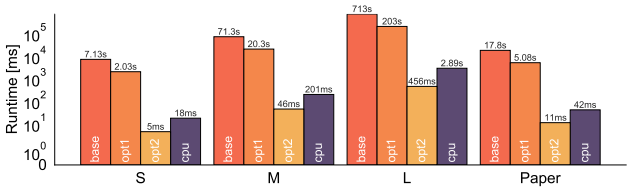

Fig. 4: Runtimes of Azimint implementations.

version becomes slower, and for the *L* benchmark, it is slower than the baseline. This can be attributed to the increasing number of memory operations that overshadow the advantages provided by the optimized dot-product.

Due to extensive use of streams, the *Input-Output Optimized* version is not synthesizable for the specified benchmark sizes. The synthesis for the *S* benchmark crashed after 66 hours. We compared the optimized implementation to the baseline implementation for an input size of 100 and obtained a speedup of **6.8x**.

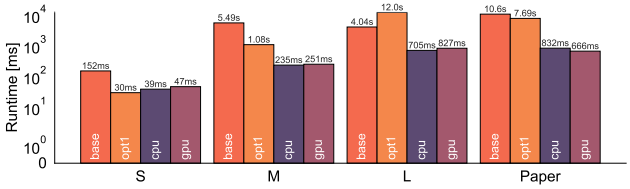

Fig. 5: Runtimes of Durbin implementations.

4.4 Gram-Schmidt

Our optimized version achieves roughly **10x** speedup compared to the baseline. However, it remains slower than the CPU implementation, as too many dependencies in the algorithm limit the available parallelism. Furthermore, the computational intensity is low, as we continuously need to access memory during the computation, which limits loop unrolling. However, our implementation significantly improves performance over the GPU.

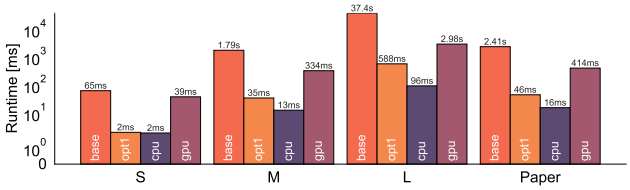

Fig. 6: Runtimes of Gram-Schmidt implementations (hardware emulation used for FPGA runtimes).

4.5 Cavity Flow

The buffered version of our implementation achieves **1.2x** speedup over the baseline implementation by reducing the number of memory accesses. However, as seen in figure 7, this is much slower than the CPU implementation. This leads us to believe that for iterative code (output of i^{th} iteration is input of $i-1^{th}$ iteration) with data dependence (section 3.5), optimizing for FPGAs is a challenging task.

In the vectorized version, the *u* and *v* matrices were vectorized. However, the speedup due to this vectorization

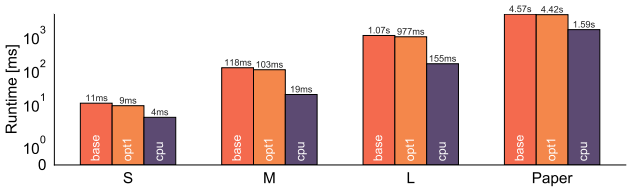

Fig. 7: Runtimes of Cavity Flow implementations.

is negligible. This is because of the dependency described in Section 3.5. We tested the code with different values for *W*: 4 and 16. Increasing the size of *DataPack* leads to an improvement in the runtime. This can be attributed to the amortization of the latency introduced by the edge cases.

4.6 Conv2D

The first optimized version implements loop reordering, while the second adds buffering, scaling, and unrolling. Due to the loop unrolling, there are significant improvements in runtime and scalability compared to the baseline. For smaller inputs, the speedup is low since the optimized version pays extra overheads for buffering. For larger inputs, the runtimes diverge significantly. Our version performs better than both CPU and GPU by **8.3x** and **10.7x** respectively. We also compile, link, and run the optimized conv2d kernel on FPGA hardware for the *M* benchmark. The resulting runtime is **70ms**, slightly higher than the value from emulation. The difference is attributed to the kernel call overhead. We conclude that the convolution kernel is well-suited for FPGAs.

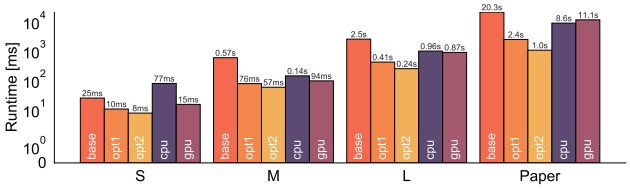

Fig. 8: Runtimes of Conv2D implementations (hardware emulation used for FPGA runtimes).

5 Conclusions

We successfully implemented optimized versions of five NPBench kernels covering a wide range of problems: *azimint*, *durbin*, *gramschmidt*, *cavityflow* and *conv2d*. We implemented and optimized the kernels using the Xilinx Vitis HLS toolchain [24], HLS compiler directives [16] and *hlslib* [17]. In all cases our optimizations were able to achieve speedups compared to the baseline. *azimint* in particular improved by **1564x**. Further, we were able to evaluate *conv2d* and *gramschmidt* kernels with hardware emulation. Finally, *azimint* achieves a **3.7x** speedup over the DaCe [6] CPU version, while *conv2d* achieves speedups of **8.3x** and **10.7x** over CPU and GPU respectively.

We conclude that azimint and conv2d kernels are suitable for FPGA (due to high levels of parallelism), while gramschmidt, durbin and cavityflow have dependency patterns which limit the available opportunities for optimization.

6 References

[1] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler, "NPBench: A Benchmarking Suite for High-Performance NumPy," in *Proceedings of the 2021 International Conference on Supercomputing (ICS'21)*, Jun. 2021.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sept. 2020.

[3] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, New York, NY, USA, 2015, LLVM '15, Association for Computing Machinery.

[4] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud, "Pythran: Enabling static optimization of scientific python programs," *Computational Science & Discovery*, vol. 8, no. 1, pp. 014001, 2015.

[5] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[6] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, SC '19.

[7] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da, "Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance," *Xilinx Whitepaper*, 2017.

[8] John W Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers, "A low-latency library in fpga hardware for high-frequency trading (hft)," in *2012 IEEE 20th annual symposium on high-performance interconnects*. IEEE, 2012, pp. 9–16.

[9] Gareth W Morris, David B Thomas, and Wayne Luk, "Fpga accelerated low-latency market data feed processing," in *2009 17th IEEE Symposium on High Performance Interconnects*. IEEE, 2009, pp. 83–89.

[10] Derek Chiou, "The microsoft catapult project," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2017, pp. 124–124.

[11] Derek Chiou, "Intel acquires altera: How will the world of fpgas be affected?," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 148–148.

[12] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu, "When fpga meets cloud: A first look at performance," *IEEE Transactions on Cloud Computing*, 2020.

[13] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler, "Productivity, portability, performance: data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[14] Yohann Uguen and Eric Petit, "Pyga: A python to fpga compiler prototype," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, New York, NY, USA, 2018, AI-SEPS 2018, p. 11–15, Association for Computing Machinery.

[15] Declan O'Loughlin, Aedan Coffey, Frank Callaly, Darren Lyons, and Fearghal Morgan, "Xilinx vivado high level synthesis: Case studies," 2014.

[16] Xilinx Inc., "SDAccel Development Environment Documentation: HLS Pragmas," 2019.

[17] Johannes de Fine Licht and Torsten Hoefler, "hlslib: Software Engineering for Hardware Design," Nov. 2019.

[18] Giannis Ashiotis, Aurore Deschildre, Zubair Nawaz, Jonathan P. Wright, Dimitrios Karkoulis, Frédéric Emmanuel Picca, and Jérôme Kieffer, "The fast azimuthal integration Python library: *pyFAI*," *Journal of Applied Crystallography*, vol. 48, no. 2, pp. 510–519, Apr 2015.

[19] J. Durbin, "The fitting of time-series models," *Revue de l'Institut International de Statistique / Review of the International Statistical Institute*, vol. 28, no. 3, pp. 233–244, 1960.

[20] Johannes de Fine Licht, "Modeling and implementing high performance programs on fpga," 2016.

[21] Samuel Williams, Andrew Waterman, and David Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[22] Martin Langhammer and Bogdan Pasca, "High-performance qr decomposition for fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 183–188.

[23] SPCL, "NPBenchInfo," `https://github.com/spcl/npbench/tree/main/bench_info`, 2021.

[24] Vinod Kathail, "Xilinx vitis unified software platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2020, FPGA '20, p. 173–174, Association for Computing Machinery.