FPGA Acceleration of Fluid-Flow Kernels

Ryan Blanchard, Greg Stitt, Herman Lam
Dept. of Electrical and Computer Engineering
University of Florida
Gainesville, Florida, USA
rlb1116@ufl.edu, gstitt@ufl.edu, hlam@ufl.edu

***Abstract*—Adding heterogeneity to high-performance systems has the potential to meet the increasing performance demands of applications beyond the era of Moore's Law speedup. In this paper we design and evaluate an FPGA pipeline to accelerate the CMT-bone-BE physics application, which is highly scalable, but performance bound by compute-heavy fluid-flow kernels. For the targeted kernels, a single instance of our FPGA pipeline shows a speedup of up to 9.4x over a CPU core. For the entire application workload, this kernel acceleration provides an average 1.5x speedup over a CPU core, while also outperforming a GPU. With even mid-range FPGAs providing enough resources to replicate our pipeline over 8 times, we project that the FPGA accelerator can add and/or replace 12 equivalent cores with a low-power alternative, with additional time-sharing optimizations potentially increasing that amount significantly.**

***Index Terms*—FPGA, Heterogeneous Computing, High-performance Computing, Hardware Accelerators**

I. Introduction

As we approach the end of Moore's Law and the "free" speedup associated with smaller and more plentiful transistors, computer architectures require more efficient and creative use of available resources to meet increasing performance demands. High-performance computing (HPC) initially addressed this need by increasing the number of available CPUs and CPU cores. However, the general-purpose nature of these cores results in prohibitive performance for many applications, or requires a number of CPUs that has prohibitive power, energy, and/or cost.

To address the limitations of general-purpose cores, HPC has started a rapidly increasing trend towards heterogeneous systems, with specialized hardware accelerators alongside CPUs. The most common hardware accelerator is the GPU, as evidenced by many of the Top 500 supercomputers adopting a heterogeneous CPU-GPU architecture [1]. Yet the GPU is not the only, or final, solution to hardware acceleration.

The anticipated trend in future HPC system development is towards that of extreme heterogeneity, where many different types of hardware accelerators will exist alongside CPUs and GPUs [2]. This trend will allow different tasks to run on hardware best suited for its needs in the interest of optimal performance and efficiency of the overall system and workload. A potentially major stepping stone in this path toward increasing heterogeneity is the field-programmable gate array (FPGA), which has been shown to have performance and especially energy advantages over GPUs for some applications [3], [4], [5]. FPGAs consist of a reconfigurable hardware fabric, which allows for the creation of computational circuitry customized to a given application. Parallelization can be achieved by deeply pipelining operations, as well as by replicating circuitry to increase computational bandwidth.

In this paper we evaluate the potential for FPGAs as hardware accelerators of fluid-flow kernels by comparing performance with a CPU and GPU to better understand the design-space options for these kernels. Fluid-flow kernels are an important part of many scientific applications studying physical phenomena in a volume [6], [7], [8]. To evaluate these kernels, we accelerate the *CMT-bone-BE* application, which serves as a proxy representation of *CMT-nek*: a scientific physics code under development at the Center for Compressible Multiphase Turbulence (CCMT) [9]. CMT-nek is an expansion of *NEK5000*, a well-established code in the field of fluid-flow physics [6], which was designed to perform incompressible fluid flows based on Navier-Stokes equations.

We show that FPGAs are an attractive option to accelerate performance of these fluid-flow kernels beyond CPU scaling due to both wide and deep parallelism that we can achieve with a custom compute pipeline. Specifically, with parallel circuitry we are able to reduce an $O(n^4)$ computational kernel to $O(n^3)$ while decreasing memory intensity. Additionally, our pipeline enables streaming data that overlaps computation with communication; by the time computation is done, most of the results are already back in the host CPU's memory.

We show that our FPGA pipeline accelerates partial-derivative computations by up to a 9.4x speedup over a CPU core, which for the the entire fluid-flow timestep, referred to as the *software workload* for simplicity, results in over 2x speedup (as limited by Amdahl's Law [10]). Although a 2x overall speedup is not sufficient to replace multi-core execution with a single FPGA pipeline, the pipeline can complement existing multi-cores. For example, one FPGA pipeline provides on average about 1.5 cores of performance, or can reduce power by using fewer total cores. Furthermore, modern FPGAs can fit multiple copies of our pipeline, with up to 8 replications potentially fitting on the targeted Arria 10 FPGA. For 8 replicated pipelines, the FPGA could potentially replace and/or add 12 cores to existing systems. Additionally, time sharing of the FPGA accelerator by multiple CPU cores

This work is supported by the U.S. Department of Energy, National Nuclear Security Administration, Advanced Simulation and Computing Program, as a Cooperative Agreement under the Predictive Science Academic Alliance Program, under Contract No. DE-NA0002378.

could allow for each pipeline to accelerate an average of 5 cores, rather than just one. So on average, 8 pipelines with time sharing could allow a single FPGA to accelerate 40 CPU cores for a total compute power of 60 core equivalents.

Despite having a lower peak computational throughput than the GPU, the average speedup of the FPGA-accelerated core was 8% better than the GPU-accelerated core for larger element sizes of 16–32, while offering lower power consumption. Meanwhile for smaller sizes, the FPGA significantly outperforms the GPU by an average 17.7x speedup for element sizes of 5–10. This performance benefit, when multiplied with lower power consumption, can yield massive energy savings over the GPU.

The remainder of this paper is organized as follows: Section II discusses related research. Section III gives an overview of our CMT-bone-BE case-study application. Section IV describes the custom FPGA pipeline. Section V compares application performance of the FPGA pipeline with the CPU and GPU baselines. Section VI concludes the paper.

II. Related Research

The prevalence of spectral-element methods for solving computational fluid dynamics and other scientific applications has lead to many corresponding acceleration studies. Here we present some of the efforts made in the context of scaling up to large homogeneous CPU systems, as well as the adaption of GPU and FPGA accelerators.

A. Homogeneous Parallelization

Fischer et al. presented theoretical analysis for strong scaling of particle-based flow simulations (including NEK5000) up to very large numbers of CPU cores [11]. While dividing large workloads among parallel compute nodes can significantly cut run time, eventually inefficiencies at very large processor counts set scaling limits. These theoretical strong-scaling limits were tested for NEK5000 by Offermans et al., where high latency and noise across networks can further reduce scaling limits by an order of magnitude or more [12].

Hutchinson et al. studied the tradeoff between computational cost and simulation accuracy for the spectral-element code NekBox on large-scale CPU systems [13]. They integrated the use of the LIBXSMM library to offer optimized performance on small matrix-multiplication operations on modern Intel CPUs, and showed improved performance and peak memory bandwidth of Nek codes.

B. Hardware Acceleration with GPU

Fischer et al. also considered the GPU in their scaling analysis, noting the extra layer of parallelism that can be obtained from within the GPU itself [11].

Otten et al. studied the performance benefits of using multiple GPUs for similar spectral-element solvers [14], but their use of GPUDirect storage allowed them to bypass much communication with the CPU. Such communication is common in acceleration environments, which we evaluate in this paper.

Markidis et al. ported NekBone, a skeleton app of NEK5000, to a multi-GPU system using OpenACC compiler directives [15]. They showed marginal performance improvement with an unoptimized initial version, but roughly double this performance by flattening the loops of the compute-heavy matrix-matrix multiplication kernel.

CCMT researchers Gadou et al. explored performance and energy tradeoffs of various workload distributions of another proxy-app for CMT-nek on CPU/GPU systems [16], [17]. They showed improved performance with multiple GPUs at the cost of increased power consumption.

C. Hardware Acceleration with FPGA

Grigoras et al. considered the performance benefits of FPGAs in the context of similar spectral finite-element methods, but they focused on the sparse matrix-vector multiplication kernel [18]. While our target operations involve 3-D by 2-D matrix multiplication, which can be conceptualized as an array of matrix-vector multiplications, the matrices in CMT-Bone-BE are very dense and cannot make use of sparsity techniques.

While FPGA implementations of spectral element and fluid flow kernels are not common, there has been work on more general matrix multiply based operations on FPGAs. Kestur et al. tested Basic Linear Algebra Subroutines (BLAS), and found performance similar to CPUs and GPUs for smaller matrix sizes with considerable energy efficiency benefits [19].

III. Application Overview

In this study we investigate speeding up CMT-bone-BE using hardware accelerators. CMT-bone-BE is representative of the computational requirements and scaling trends of its parent application, CMT-nek, but at a more coarse-grained level, which allows us to study its behavior with less complexity [20]. CMT-nek adds three important characteristics to the Navier-Stokes-based incompressible fluid flows performed by NEK5000, namely *Compressibility* of flows, *Multiphase* states of matter (e.g., simulation of solid particles in addition to gaseous fluid flows), and *Turbulence* related to high pressure disturbances, such as explosions in particular.

A. Algorithm Summary

CMT-nek performs calculations of fluid flow on a large 3-D volume over small timesteps, and divides the space into a grid of many small 3-D elements. The bulk of execution time comes from partial-derivative calculations on each of these 3-D elements and communication of flow between neighboring elements. CMT-bone-BE captures this behavior as the basis for its coarse-grained abstraction of the parent application. Each simulation timestep involves the calculation of the partial derivative fluid-flow kernels in three directions for five different physical parameters and three Runge-Kutta stages for every element in the physical volume under simulation. These derivative calculations, outlined in Algorithm 1, are essentially a specialized matrix multiplication between 3-D matrices (or *ternices*) representing the element data, and 2-D matrices representing the corresponding derivative operations. From

Algorithm 1 Partial Derivative Compute Kernel

```
1: for derivative = dr, ds, dt do
2:     for i = 1, ..., N do
3:         for j = 1, ..., N do
4:             for k = 1, ..., N do
5:                 for g = 1, ..., N do
6:                     if dr then
7:                         C_r[i][j][k] += A[i][g] * B[g][j][k]
8:                     end if
9:                     if ds then
10:                        C_s[i][j][k] += A[j][g] * B[i][g][k]
11:                    end if
12:                    if dt then
13:                        C_t[i][j][k] += A[k][g] * B[i][j][g]
14:                    end if
15:                end for
16:            end for
17:        end for
18:    end for
19: end for
```

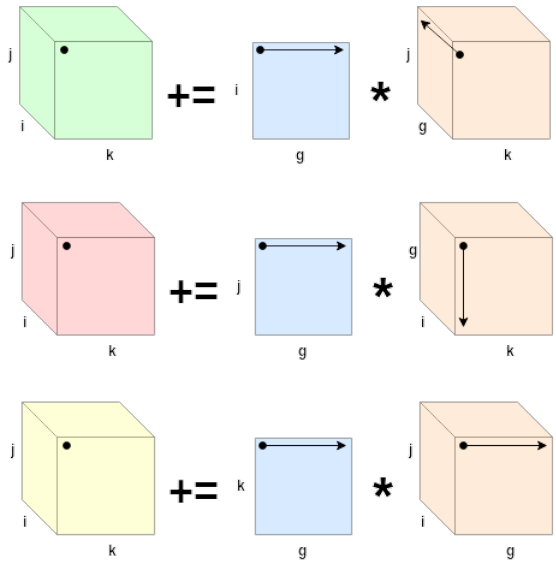

Fig. 1. Example 3-D by 2-D matrix multiply showing calculation of one output item for each partial derivative operation (dr, ds, dt), which traverse the input ternix in different directions.

NEK5000 down to CMT-bone-BE these derivative calculations operate on the order of N^4 with storage requirements on the order of N^3, where N refers to the grid size of each element, which is stored as a NxNxN matrix.

B. Target Kernel Operations

The specialized matrix multiplication for partial derivatives is the focus of our acceleration efforts due to its computational complexity, as well as its ability to be decomposed into many discrete operations. An example visualization of our 3-D matrix multiply operations can be seen in Fig. 1. Here the 2-D NxN matrix has fixed contents for each derivative operation kernel; we will refer to this matrix as **A**. The input 3-D NxNxN matrix (right side) holds the data associated with the fluid volume of the current element and is multiplied by the kernel matrix **A**; we will refer to this matrix as **B**. The output 3-D NxNxN matrix (left side) holds the results of the accumulated matrix multiplications between **A** and **B**; we will refer to this matrix as **C**.

There are three different outputs (shown by different colors) corresponding to the three directions of flow (*dr, ds, dt*) for which each input element has a derivative calculated. As shown in the figure, each element of the output matrix **C** (represented by a single dot) is the sum of an entire N-length vector row of **A** multiplied element wise with an N-length vector of **B** and accumulated into a single output. Since there are N^3 items in the output matrix, and it takes N multiplications to produce each output, it is easy to see that the entire matrix multiplication is of the order N^4 operations. The main difference between the three directions of flow of the derivative operations (*dr, ds, dt*) is the order in which the input matrix **B** is traversed and the corresponding items it is multiplied with in **A**. This presents a challenge for our custom pipeline to be able to handle processing the same input data in three different orders to produce three different results.

Serial execution of a full partial derivative kernel, as shown in Algorithm 1, involves a four-way nested loop, in which the innermost loop iterates through index *g* to perform the N multiply-accumulates required for each output point in **C**. The loops then iterate through *i*, *j* and *k* to calculate the results for every point in each output matrix **C**. Since each output point is calculated independently of every other output point, there are no output dependencies and the computations can be done in parallel. Hence, it is within these operations that we will attempt to extract speedup from a fine-grained parallelization.

C. Parallelization Strategies

NEK5000 has been shown to scale well across a large number of parallel processes in an MPI-based environment [12]. By extension, CMT-nek and CMT-bone-BE also scale well. However, this parallelization is achieved splitting the entire large 3-D volume space into many small 3-D elements distributed across MPI ranks. By constrast, we are interested in extracting more fine-grained parallelism from within the work distributed to each core, in order to supplement the existing scalability. By comparing an FPGA, GPU, and single CPU core, we are finding the most effective way to perform the underlying calculations of CMT-bone-BE—a fundamental principle in striving towards extreme heterogeneity.

Of course, the performance of the accelerators in the context of a many-core CPU environment is still important, which we evaluate with projections of the equivalent number of cores that can be added and/or replaced by each accelerator. In other words, we do not intend for the accelerators to replace the multi-core, and instead evaluate how they complement each other in different potential system architectures.

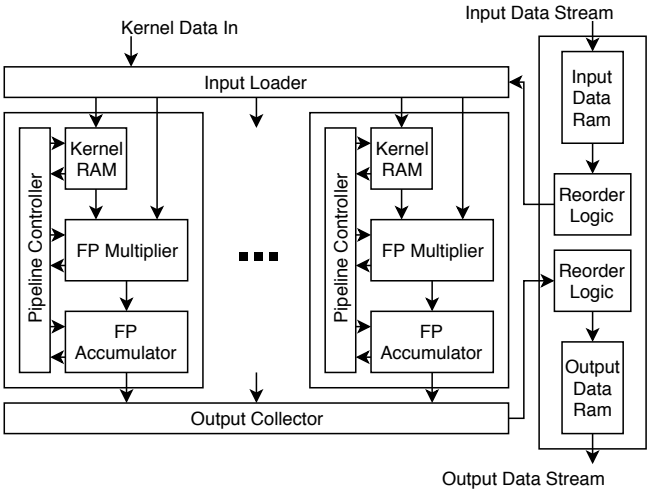

Fig. 2. High-level block diagram of FPGA partial-derivative accelerator with input and output reorder buffer.

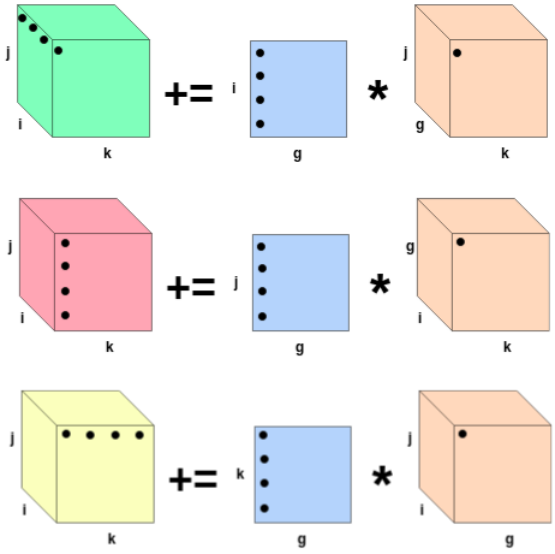

Fig. 3. Example reorder of matrix multiply for FPGA. One input simultaneously multiplied by N kernel items to accumulate into N different partial sums in different orders for each derivative direction.

IV. FPGA Acceleration Architecture

A high-level overview of the FPGA accelerator architecture for the partial-derivation computations of CMT-bone-BE (Algorithm 1) is shown in Fig. 2.

In trying to accelerate these special matrix-multiply kernels on an FPGA, we seek to take advantage of replicating multiply and accumulate circuitry to perform multiple operations in parallel, as well as using a deep pipeline to allow us to stream inputs and produce outputs on *every* clock cycle once the pipeline is filled. A major bottleneck of hardware accelerators is communication bandwidth, so we would like to minimize how much data has to be sent to the FPGA. In order to minimize data streaming, we need to rearrange the operations of the kernel to make it more efficient on the FPGA.

Looking at the execution of this kernel we see that there are N^3 inputs and outputs, with N^4 multiplies and accumulates. This implies that each input element is needed for N calculations of N different outputs, which means a naive implementation on the FPGA would have to stream each input N times. Rather, if we perform all calculations with a given input element at the same time, then we only have to stream each input once, thus reducing the memory intensity of the workload. This concept is illustrated in Fig. 3, where each input element is multiplied by N different items in the 2-D kernel matrix, which then accumulate into N different partial sums of the output ternix. If we stream the input elements in the correct order for each derivative, the N partial sums will fully accumulate after N inputs, and then move on to the next set of partial sums, thus producing an average of one complete output every clock cycle. This streaming order is important, and also different for each direction of the derivative calculations, which we will address shortly.

The data pipeline design implements N parallel datapaths which each correspond to one row of the NxN kernel matrix. During initialization, the NxN kernel is loaded into small block RAMs in each datapath; only the N items within each row of the NxN kernel are stored in a given datapath. Then input data is streamed one item per cycle into every datapath and is multiplied by N different items of the kernel RAMs, one on each path. These accumulate into N different partial sums, and after every N accumulations a full sum is completed in each datapath. These N concurrent results are collected and shifted out in sequential order while the next N partial sums are accumulating. Overall, this strategy allows for one result to be produced every clock cycle after the pipeline is filled.

As mentioned, the streaming order of the input data distinguishes the three different derivative directions. All three derivative directions use the same input data for a given element, but they use the data in a different order to capture the directional flow. Only one of these three derivatives can stream the data through the given pipeline in order of memory storage, based on C-wise row-major array ordering. The other two derivative directions require column and depth-wise streaming of the 3-D input array data. This streaming is handled by a reorder buffer wrapped around the multiply accumulate pipeline. This reorder buffer stores input items in a block RAM and sends the data in the necessary order into the pipeline by first stepping through it with a step size of 1 for derivative direction *dt* (e.g., the natural storage/streaming order coming from memory), followed by a step size of N for derivative direction *ds*, and finally a step size of N^2 for derivative direction *dr*. Accumulated results stream out from the pipeline in the same order that they stream in, but are stored into another block RAM in the original row-major memory order by another reorder buffer. These outputs can then be streamed in order back to the CPU host memory.

TABLE I
EXECUTION TIME AND SPEEDUP OF TARGET KERNEL

Target Kernel Speedup vs CPU					
Element Size	CPU Time (ms)	GPU Time (ms)	GPU Speedup	FPGA Time (ms)	FPGA Speedup
5	0.003	0.279	0.01	0.009	0.35
8	0.017	0.423	0.04	0.015	1.15
10	0.041	0.516	0.08	0.023	1.74
12	0.082	0.034	2.41	0.036	2.30
16	0.263	0.068	3.88	0.077	3.39
20	0.659	0.101	6.53	0.148	4.46
25	1.710	0.171	10.01	0.281	6.09
32	5.575	0.355	15.68	0.594	9.38

TABLE II
EXECUTION TIME AND SPEEDUP OF TOTAL TIMESTEP

Total Timestep Speedup vs CPU					
Element Size	CPU Time (ms)	GPU Time (ms)	GPU Speedup	FPGA Time (ms)	FPGA Speedup
5	0.008	0.379	0.02	0.013	0.59
8	0.039	0.531	0.07	0.037	1.06
10	0.085	0.642	0.13	0.068	1.25
12	0.178	0.234	0.76	0.134	1.33
16	0.571	0.508	1.12	0.393	1.45
20	1.360	0.931	1.46	0.869	1.56
25	3.130	1.748	1.79	1.758	1.78
32	9.203	4.318	2.13	4.421	2.08

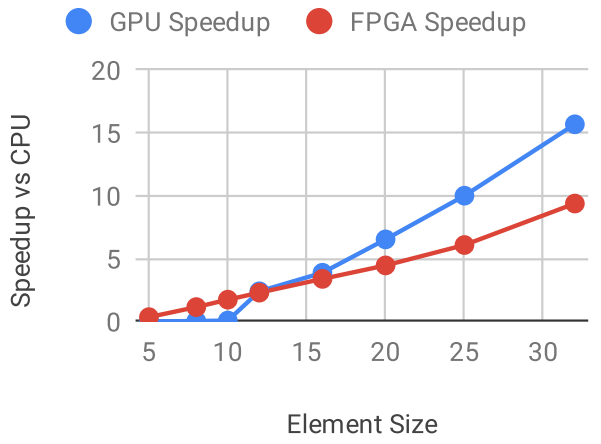

Fig. 4. Speedup for GPU and FPGA relative to a CPU core for the targeted compute kernel.

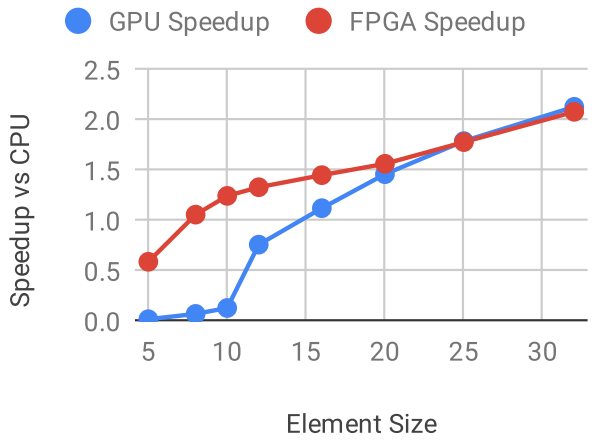

Fig. 5. FPGA and GPU accelerated core speedup relative to non-accelrated CPU core for an entire CMT-bone-BE timestep including initialization and cleanup.

V. RESULTS

A. Experimental Methodology and Setup

In this study we are investigating methods to go beyond the performance gain of CPU scaling by extracting additional fine-grained parallelism that is inherent within the tasks of the workload. In order to show this, we need to look specifically at the work that each individual CPU core would perform. In our case, with CMT-bone-BE, each CPU core will perform all of the tasks of the application for each simulated timestep, but only for a subsection of elements in the overall problem volume. Of these tasks, the major computational bottleneck is in performing the special matrix-multiplication kernels used in calculating the derivatives of flow. While there is significant parallelism within these individual matrix multiplications, the CPU core is not able to fully expose it. This is due to the fact that the overall workload has already been distributed amongst all of the available cores, so there are no cores left to further divide these calculations. Further, even if we had more cores available, spreading these fine-grained operations among non-specialized CPU cores could only achieve linear speedup at best, although realistically the increased communication and memory overheads would negate a significant portion of the benefit. Therefore, it makes sense for our methodology to measure a single CPU core executing its workload as a baseline reference, as this is the expected behavior of each core in a system. We can then project how the FPGA and GPU can complement a multi-core system in terms of numbers of equivalent cores added or replaced.

To get the CPU core baseline, CMT-bone-BE was modified to run on a single CPU core, as opposed to its normal operation of being distributed among MPI ranks. For the GPU baseline, the same code was run on the CPU host, except that the targeted matrix-multiply kernels were offloaded to execute on the GPU accelerator. This is the same use case for the FPGA accelerator, where only the targeted kernels are executed by the FPGA, and everything else runs on the CPU host.

Performance was measured by software timers in the CPU code which captured time spent on the matrix-multiply kernels and the total timestep, as well as memory initialization and cleanup. The total timestep measurements included many iterations of the partial derivative kernels, as well as several other less expensive operations which make up the rest of the

application behavior. By looking at the performance of the total timestep, we were able to see how accelerating the partial derivatives improved the overall application performance.

We collected results for CMT-bone-BE execution times for CPU and GPU on the HiPerGator supercomputer at the University of Florida [21]. The CPU used to measure performance for all non-accelerated tasks was the Intel Xeon E5-2698v3 processor. The GPU accelerator used was the NVIDIA Tesla K80. The GPU implementation used CUDA to offload the targeted compute kernel across the GPU's parallel compute cores. Both the CPU and GPU codes were compiled with the nvcc compiler with O2 level optimizations. The FPGA implementation was run on the Intel DevCloud [22], which used an Intel Arria 10 GX FPGA 10AX115N2F40E2LG. The custom compute pipeline was implemented in RTL with a combination of VHDL and SystemVerilog. This was synthesized on the DevCloud with Quartus Prime Pro version 19.2.0. In the interest of maintaining a single CPU baseline metric, and due to the lack of a common system with both an FPGA and GPU, the total timestep results for the FPGA use the CPU measurements from HiPerGator, with measured run time of the FPGA kernel (including communication costs of transferring the input kernel, data and output results) from the DevCloud substituted in place of the CPU kernel time. As future work, we are looking into finding an HPC system with both an FPGA and GPU.

The primary parameter which we are varying in this study is the element size, which corresponds to N of our NxNxN data matrices. Typical operation of the parent application CMT-nek uses element sizes in the range of 5–25. In order to accommodate this, the FPGA implementation was developed to support up to an element size of 32, and we measured performance for our three implementations for element sizes from 5–32. This range scales the total workload over 250x, which allows us to see potential tradeoffs for the hardware accelerators on large and small input sizes.

B. Accelerator Comparison

The targeted compute kernel experienced significant speedup over a single core from both the FPGA and GPU accelerators. As seen in Fig. 4 and Table I, both the FPGA and GPU see a large increase in speedup of the targeted kernel execution times as the element size increases. The nature of the FPGA in producing an output every clock cycle once its pipeline is filled allows it to take advantage of this increasing element size, as it deepens the pipeline and increases its operation-level parallelism. Similarly, the GPU can utilize more of its compute cores in parallel when there is more data available to operate on at a given time.

Interestingly, the GPU actually sees a slowdown compared to the CPU in the cases of element size 5–10. Looking at the target kernel operation time in Table I, we can see that the GPU time for a kernel of element size 5 is greater than that of size 25, while element sizes of 8 and 10 are both larger than that of size 32. We also see an order of magnitude reduction in time going from size 10 to 12. It is unsure why the magnitude of these performance differences is so large and sudden for the smaller element sizes, although in general the trend may be attributed to overhead associated with tiling and shared memory in the GPU far outweighing the benefits of parallelization for such small problem sizes.

TABLE III
TIME MEASUREMENTS FOR CPU

CPU Execution Time (s)					
Element Size	Memory Allocation	Kernel Total	Timestep (other)	Memory Cleanup	Total Time (s)
5	0.0016	0.003	0.003	0.0003	0.008
8	0.0054	0.017	0.016	0.0007	0.039
10	0.0097	0.039	0.035	0.0011	0.085
12	0.0160	0.079	0.082	0.0016	0.178
16	0.0355	0.252	0.278	0.0054	0.571
20	0.0650	0.633	0.650	0.0128	1.360
25	0.1208	1.642	1.344	0.0235	3.130
32	0.2504	5.352	3.560	0.0403	9.203

The FPGA also experienced a slowdown versus the CPU, but only for element size 5. Here, the latency associated with accessing the host CPU's shared memory over PCIe was costlier than the amount of parallelism which could be extracted from such a small problem size. Despite this, the FPGA significantly outperforms the GPU accelerator for this size, and actually averages over a 27x speedup versus the GPU for element sizes 5–10. An advantage the FPGA had over the GPU was overlapping the overhead of data transfer with computation, which is especially important for smaller problems with less computation available to amortize communication costs. The FPGA could begin processing (and even return results) as the input data was still being streamed in. Conversely, the GPU had to wait until all of the input data was finished transferring before it could begin operation. However, the GPU does provide a higher peak computational throughput than the FPGA, which is evidenced by the higher performance on larger target kernel sizes.

The acceleration of the target kernel also resulted in speedup of the overall application, as shown in Table II and Fig. 5. However, the overall speedups are limited to more modest numbers by Amdahl's Law, due to the remaining CPU execution of the code which was not accelerated. The FPGA showed steady increases in speedup with increasing element size up to over a 2x speedup of the overall workload, and an average speedup of 1.4x. This limit around 2x speedup makes sense when looking at the breakdown of execution times for the CPU operation in Table III. Here the time spent on the targeted compute kernel takes up roughly 40–60% of the total workload time. This leaves about half of the total execution time still needing to be run by the CPU, even if the accelerators were able to reduce the targeted kernel time down to nothing.

Fig. 5 shows the speedup of FPGA and GPU accelerated cores versus non-accelerated cores on the overall workload. The FPGA and GPU accelerators produce roughly the same

performance benefit at the largest sizes, despite the GPU's clear advantage on the target kernel. This is due to a larger cost of allocating GPU memory. The FPGA-accelerated core outperforms the GPU accelerated core for all other test cases, and by a wide margin for sizes 5–10, where on average performance is nearly 18x better. This discrepancy at small sizes is more so due to the GPU's poor performance than the FPGA's dominance, as the FPGA also sees its worst performance relative to the CPU for these sizes. Still, this contrast in the two accelerators' performance is important. It would not make sense to deploy a hardware accelerator that slows down performance of what it is supposed to be speeding up. The GPU results in a slow-down versus the CPU for half of our test cases, and produces a 0.94x speedup across all sizes, meaning less performance than a non-accelerated core on average. Therefore, the GPU is not a very viable accelerator for this problem. Meanwhile, although the FPGA does not demonstrate its best performance at small sizes, it still shows speedup over the CPU in all but one test case, and provides an average speedup of 1.4x over a non-accelerated core. This speedup makes the FPGA a viable hardware accelerator for most use cases of this application, and demonstrates its versatility as an accelerator compared to the GPU.

Another important consideration for hardware accelerators is energy efficiency. While we have not yet explicitly measured power consumption for the accelerators running CMT-bone-BE, other studies have shown that FPGAs can operate with significantly lower power consumption than GPUs [3]. With that general trend in mind, if the FPGA is roughly equivalent to the GPU in terms of performance on the larger element sizes, then it would be more energy efficient as an accelerator at these sizes. For smaller sizes, where the FPGA outperforms the GPU by up to 30x, less power consumption would stack with reduced run time (E = P*t) to potentially see massive energy savings for the FPGA compared to the GPU.

C. Projected Acceleration Scaling

Now that we have seen the FPGA can serve as an effective and efficient accelerator of a single CPU core for this application, we will consider the potential benefits for an FPGA accelerator deployed in a multi-core system, and project how this acceleration could scale up to heterogeneous HPC. As future work, we plan to validate these projections with actual implementations on the Intel DevCloud. Such implementations were not available at the time of publication due to the significant complexities of interfacing a single FPGA with numerous CPU cores simultaneously.

We have shown that by offloading the computationally heavy portion of the workload to an FPGA accelerator, an FPGA-accelerated CPU core is able to execute its software workload up to over 2x faster, and 1.5x faster on average for cases when using an accelerator makes sense. This means that an FPGA-accelerated CPU core could perform on average 1.5x the work of a non-accelerated core in the same amount of time.

All discussion thus far has been under the premise of a single copy of our compute pipeline residing on the FPGA accelerator. However, with modern FPGA devices we can fit multiple copies of this pipeline on a single FPGA, thereby extending the possible compute power that an FPGA accelerator could supply. Initial resource requirements of our compute pipeline are roughly 27k ALM, 4 Mb of Memory and 100 DSP blocks. The Arria 10 GX 1150 offers up to 427k ALM, 67 Mb of Memory and 1518 DSP blocks [23], which is roughly 15x more resources than our pipeline in each category. We only have access to about 75% of these resources in the partial reconfiguration (PR) region of the FPGA that our accelerator design is confined to operating in, which lowers available resources to roughly 11x that of a single compute pipeline. Considering additional resource overheads of adding parallel pipelines, it appears reasonable that we could fit 8 copies of our compute pipeline onto a single Arria 10 FPGA, which we will evaluate in future work. In theory, this would produce an FPGA accelerator with an average of 12 CPU cores worth of performance, with a lower power requirement than an equivalent 12 core system.

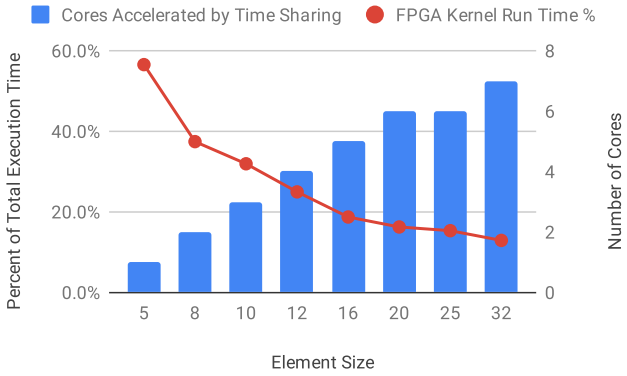

Fig. 6. Percent of total execution time taken by accelerated kernel on FPGA, and corresponding number of CPU cores that could share FPGA availability without competition.

The scaling limits of our FPGA accelerator are actually more constrained by the communication bandwidth to the device, than by available resources within the configurable device. For full performance, each compute pipeline requires an average of 64 bits (one floating point double data type) of input and output communication bandwidth every clock cycle of its operation. While we may not currently be able to fully supply 8 compute pipelines on the FPGA with this amount of data simultaneously, future developments in this area, such as the emerging PCIe5, will significantly increase communication bandwidths.

Fortunately, we may be able to have more CPU cores use the accelerator without increasing the communication bandwidth. Since the CPU only uses the FPGA to accelerate the targeted compute kernel, and because there is a non-neglible amount of software workload remaining for the CPU core (i.e., around 50% of the original non-accelerated execution time), the FPGA actually spends much of its time idle if only accelerating a single core. If we are able to effectively stagger requests to the

FPGA, we could have more cores make use of the accelerator by sharing it at different times. For element size of 32, the FPGA is in use calculating partial derivatives only 12.9% of the total run time. This means that 87% of the time, the FPGA could be used by other cores. If all the CPU cores are able to stagger their use of the FPGA, and perform their remaining software workload when not using it, then a single FPGA compute pipeline could support up to 7 different CPU cores without competition for FPGA availability.

Fig. 6 shows the number of CPU cores that could be accelerated by sharing the availability of the FPGA, based on the percentage of total run time that the FPGA requires to complete each compute kernel. On average across our viable test cases, 5 CPU cores could share an FPGA pipeline cooperatively. With 5 cores being accelerated to 1.5x average performance, our FPGA accelerator could now facilitate roughly 7.5 CPU cores worth of computational power with a single compute pipeline. In theory, if communication bandwidth is increased to be able to supply 8 compute pipelines simultaneously, and time sharing of each of these pipelines is employed, then a single Arria 10 FPGA could on average accelerate 40 CPU cores to achieve the computational power of 60 cores. For the best case element size of 32, this could be 56 cores accelerated, producing approximately 116 cores worth of compute power (i.e., 60 low-power core equivalents added) with a single FPGA accelerator. While this is may not be fully attainable with today's systems, we believe it showcases the potential for FPGA accelerators in future heterogeneous HPC systems.

VI. Conclusion

In this paper we evaluated the viability and versatility of FPGAs as hardware accelerators in future HPC systems as we trend towards extreme heterogeneity. To do so, we performed a case study on the fluid-flow kernels of the spectral-element solver CMT-nek. By focusing on the portion of work that is performed by each individual core of a large system running a highly scalable application, we were able to test whether hardware accelerators can extract a finer level of parallelism from the workload than coarse-grained CPU scaling alone.

We designed and implemented a custom compute pipeline for the fluid-flow kernels on an FPGA, and measured its performance relative to both a CPU and a GPU baseline. Our experiments show that the FPGA is able to run the targeted compute kernel 3.6x faster than a CPU core on average. By accelerating these computationally heavy fluid-flow kernels, the FPGA can speed up the overall workload over 2x of a CPU core, with an average improvement for viable cases of 1.5x. Our results show the FPGA as a sensible hardware accelerator for this application, as it is able to speed up the overall workload in all but one use case size. The GPU shows comparable performance to the FPGA at larger input sizes, but suffers significant slowdown versus the FPGA and even the CPU at smaller sizes, making it unusable as a hardware accelerator for those problem sizes.

We will seek to show the capabilities of our FPGA design in the context of accelerating a multi-core CPU system in future work. In a notional heterogeneous HPC system, if we are given one FPGA accelerator per X number of CPU cores, we would like to study ways to potentially optimize the configuration of this accelerator. This will involve adding multiple copies of the compute pipeline on the FPGA, communicating to the FPGA with multiple CPU cores, and time sharing of the compute pipelines by multiple cores.

While we have not yet measured power consumption for these devices running this application, we do outline a case for the FPGA being a significantly more energy efficient hardware accelerator compared to the GPU for this application. Properly demonstrating this advantage in energy efficiency of the FPGA accelerator will be part of our future work on this problem.

References

[1] E. Strohmaier, H. Simon, J. Dongarra, and M. Meuer, "Japan captures top500 crown with arm-powered supercomputer," Jun 2020. [Online]. Available: https://top500.org/news/japan-captures-top500-crown-arm-powered-supercomputer/

[2] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf *et al.*, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity," 12 2018.

[3] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. Baylis, "Scalable window generation for the intel broadwell+arria 10 and high-bandwidth fpga systems," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, p. 173–182. [Online]. Available: https://doi.org/10.1145/3174243.3174262

[4] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, p. 326–331. [Online]. Available: https://doi.org/10.1145/2934583.2934644

[5] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, "A 7.663-tops 8.2-w energy-efficient fpga accelerator for binary convolutional neural networks," in *FPGA*, 2017, pp. 290–291.

[6] *NEK5000*, Argonne National Laboratory, Illinois. [Online]. Available: https://nek5000.mcs.anl.gov

[7] *NekCEM*, Argonne National Laboratory, Illinois. [Online]. Available: https://nekcem.mcs.anl.gov/

[8] *NekLBM*, Argonne National Laboratory, Illinois. [Online]. Available: https://neklbm.mcs.anl.gov/

[9] *Center for Compressible Multiphase Turbulence*. [Online]. Available: https://www.eng.ufl.edu/ccmt/

[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the 1967, Spring Joint Computer Conference*, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[11] P. F. Fischer, K. Heisey, and M. Min, *Scaling Limits for PDE-Based Simulation (Invited)*, 2015. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2015-3049

[12] N. Offermans, O. Marin, M. Schanen, J. Gong, P. F. Fischer, P. Schlatter, A. Obabko, A. Peplinski, M. Hutchinson, and E. Merzari, "On the strong scaling of the spectral element solver nek5000 on petascale systems," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1706.02970

[13] M. Hutchinson, A. Heinecke, H. Pabst, G. Henry, M. Parsani, and D. Keyes, "Efficiency of high order spectral element methods on petascale architectures," in *High Performance Computing*. Springer International Publishing, 2016, pp. 449–466.

[14] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min, "An mpi/openacc implementation of a high-order electromagnetics solver with gpudirect communication," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 320–334, 2016. [Online]. Available: https://doi.org/10.1177/1094342015626584

[15] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer, "Openacc acceleration of the nek5000 spectral element code," *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 311–319, 2015. [Online]. Available: https://doi.org/10.1177/1094342015576846

[16] M. Gadou, T. Banerjee, and S. Ranka, "Multi-objective optimization of cmt-bone on hybrid processors," in *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, 2016, pp. 1–8.

[17] M. Gadou, T. Banerjee, M. Arunachalam, and S. Ranka, "Multiobjective evaluation and optimization of cmt-bone on multiple cpu/gpu systems," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 259 – 271, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210537917300689

[18] P. Grigoraş, P. Burovskiy, W. Luk, and S. Sherwin, "Optimising sparse matrix vector multiplication for large scale fem problems on fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[19] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI*. IEEE, 2010, pp. 288–293.

[20] N. Kumar, M. Sringarpure, T. Banerjee, J. Hackl, S. Balachandar, H. Lam, A. George, and S. Ranka, "Cmt-bone: A mini-app for compressible multiphase turbulence simulation software," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 785–792.

[21] *HiPerGator*. [Online]. Available: www.rc.ufl.edu/services/hipergator

[22] *Intel DevCloud*. [Online]. Available: https://devcloud.intel.com/edge/

[23] Intel, *Intel® Arria® 10 Device Overview*. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf