UC RIVERSIDE UNIVERSITY OF CALIFORNIA

Bourns College of Engineering
Department of Computer Science and Engineering
351 Winston Chung Hall Riverside,
CA 92521-0144

Technical Report UCR-CSE-2014-04001

SpVM Acceleration with Latency Masking Threads on FPGAs

Robert J. Halstead, Walid A. Najjar and Omar Huseini
Computer Science & Engineering
UC Riverside
Riverside, CA 92521
(rhalstea, najjar, ohuss001)@cs.ucr.edu

Long memory latencies are mitigated through the use of large cache hierarchies in multi-core architectures, SIMD execution in GPU architectures and streaming of data in FPGA-based accelerators. However, none of these approaches benefits irregular applications that exhibit no locality and rely on extensive pointer de-referencing for data accesses. By masking the memory latency, multi-threaded execution has been demonstrated to deal effectively with such applications.

In the MT-FPGA model a multi-threaded engine is implemented on the FPGA accelerator specifically for the masking on the memory latency in the execution of irregular applications: following a memory access, the execution is switched to a ready thread while the suspended threads wait for the return of the requested data value from memory. The multi-threaded engine is automatically generated, from C code, by the CHAT compilation tool and is customized to the specific application.

In this paper we use the Sparse Vector Matrix application to evaluate the performance of the MT-FPGA execution and compare it to the latest GPU architectures over a wide range of benchmarks.

This work was supported in part by NSF Awards CCF-1219180 and IIS-1161997

1. INTRODUCTION

The growing speed gap between processors and memory is having a tremendous impact on the design of modern processors. In today's modern multi-core architectures, roughly 80% of the chip area is devoted exclusively to the memory hierarchy in the form of L1, L2, L3 or L4 caches. Such large cache hierarchy is often a limiting factor on the number of cores that can be accommodated on a single chip. Furthermore, it is also one of the major sources of power generation on the chip.

Hardware accelerators, such as FPGAs and GPUs, deal differently with long memory latencies. FPGA-based accelerators rely, mostly, on streamed data; hence the latency cost is paid once for the first element. Modern GPUs rely on both caches and spatial locality in the form of SIMD or vector execution.

Multithreaded architectures are designed to mask rather than mitigate long memory latencies. Latency masking is done by switching the execution to another ready thread. The switching can be synchronous, occurring at fixed intervals such as every cycle, or asynchronous, ocurring whenever a long latency operation is performed. Examples of such architectures include the HEP [31], the Horizon [28; 34], the Tera MTA [9; 8], later the Cray XMT [10], the SUN UltraSPARC T1 and T2 processors [14; 27; 13]. As a corollary of Little's Law, the concurrency in multithreaded architectures is equal to the number of outstanding memory requests. Hence, on systems that support a high-throughput memory subsystem the achieved parallelism can be quite large.

In FPGA-based code accelerators a frequently executed code segment, typically a loop nest, is expressed as a hardware data path through which data values are streamed. By customizing the data path to that specific application, these structures have been shown to achieve very large speedup. Regular applications are typically targets for FPGA acceleration because data can be streamed in mass to the hardware without intervening memory offloading. Two common features of such applications are high spatial (streaming data) and temporal (sliding window) locality. C to HDL tools like ImpulseC [2], ROCCC [35], and others have historically targeted this class of applications. However, this streaming paradigm implicitly assumes the existence of some form of locality within the stream since the on-chip memory on most FPGA devices is very limited. The conventional wisdom therefore dictates that *pointer intensive, or sparse applications are not well suited* for FPGA-based code acceleration.

Because of their poor spatial and temporal locality, irregular applications pose a serious challenge to high performance computing: long memory latencies defeat any potential gains from a parallel execution. However, such applications have been shown to greatly benefit from multithreaded execution [10].

The multithreaded FPGA (MT-FPGA) [26; 24] is an execution model relying on latency masking multithreaded engines implemented on FPGA and customized to the application at hand. It relies on the same execution model as traditional multithreaded architectures: suspending a thread following a memory access, resuming a thread after its data becomes available. However, unlike previous hardware multithreading approaches, MT-FPGA does not have dedicated hardware contexts for each thread or a fixed data path where the threads are executed. Rather, the number of threads, their size, the configuration of the state of each thread and the architecture of the data path are determined at compile time and are customized to these specific threads. Within one application it is conceivable to have multiple types of threads with a different data path for each type. MT-FPGA tightly integrates the generation of latency-masking hardware threads with array and loop transformations aimed at maximizing parallelism for the target application. This allows MT-FPGA to achieve a very large number of concurrent threads (in the 1,000s) and thus greatly increase throughput/parallelism. The operation and capabilities of MT-FPGA were demonstrated on sparse linear algebra benchmarks implemented on the Convey Computers HC-2ex [24], as well as in bioinformatics algorithms[20; 21].

The work reported in [24] focused on the performance evaluation of MT-FPGA as compared to other FPGA implementations of SpMV [29] and to the multicore architectures reported in [36]. In [36], [24] and [29] the Compressed Row Storage (CRS) model was used as it is the most efficient and hence has the lowest memory bandwidth requirement. Furthermore, all three studies focused exclusively on small to medium sized matrices (under 200 MB in CRS).

In this paper we extend the MT-FPGA performance evaluation to larger matrices (larger than 1 GB in CRS) and compare it to modern GPU architectures (Nvidia Tesla C1060 and K20) with alternative storage models that better fit GPU execution. The results show that when using the CSR storage model, the MT-FPGA implemented on the Convey HC-2ex, clocked at 150 MHz with 76.8 MB/s memory bandwidth, performs as well or better than the faster clocked GPU architectures and with higher memory bandwidth.

Most notably, on very large matrices, it achieves nearly three times the throughput of the K20 (3.06 versus 1.03 DP GFLOPS). Some of the alternative sparse matrix storage models do boost the throughput of the Tesla architectures higher than what can be achieved on the HC-2ex.

The rest of this paper is organized as follows: Section 2 provides a background about software multi-threaded, GPU, and FPGA architectures as well as a discussion about sparse matrix formats. Section 3 discusses current C to HDL tools and how irregular applications with dynamic workloads can be compiled. Section 4 describes the CHAT compiler tool and Section 5 shows how the it can be used to generate a custom SpMV kernel for the Convey HC-2ex architecture. Section 6 describes our experiments, and their results. Finally the conclusions are given in Section 7.

2. BACKGROUND

2.1 Convey HC Architecture

Convey Computers [12] built the first heterogeneous FPGA machines with a shared cache coherent virtual memory space between software (CPU execution) and hardware (FPGA execution). The base HC machines, HC-1 and HC-2, use four Virtex-5 LX330 FPGAs as co-processors while the HC-1ex and HC-2ex machines use four Virtex-6 LX760 FPGAs. However, their significant feature is their shared global memory space [15; 16]. Memory allocated by the host processor can be directly accessed from the FPGA. Memory space is divided between the CPU processors, and the FPGA co-processor, and performance could depend on where the memory is allocated. With 128 GB per region, it is large enough that the CPUs can setup new jobs while the FPGA finishes processing others. Together each of the 4 FPGAs, called Application Engines (AEs), compose the co-processor. Each AE interfaces to eight Memory Controllers (MCs) via a full crossbar, which supports memory request reordering. Each MC has 2 ports (identified as even or odd) that can read or write eight bytes per cycle at 150 MHz [17]. Thus each AE has 16 memory ports that deliver a peak bandwidth of 19.2 GB/s. The co-processor's peak bandwidth is about 77.8 GB/s.

2.2 GPU Architecture

Graphics Processing Units (GPUs) have become common place in the HPC community. They consist of hundreds to thousands of simple processors executing in parallel. Originally designed as high throughput devices for image processing, they excel at SIMD applications. Coupling the high number of processors with very large memory bandwidth allows GPUs to achieve orders of magnitude speedup over general purpose multi-core CPUs.

At its core the GPU consists of streaming processors (SPs) that are coupled together with floating-point units and a shared memory cache that together to make a streaming multi-processor (SM). When running every SP, within a SM, must execute an instruction from the same memory block. Threads are grouped into warps that are assigned to individual SMs. Context switching between threads in the same warp allow a better utilization of the SPs, and help mask some of the memory latency.

On the Nvidia Kepler architecture the GPU's memory bandwidth is over 200 GB/s. However, its efficiency is correlated to how well the requests can be coalesced. This makes the GPU very effective on regular applications where each thread requests sections from a sequential block of memory. However, with irregular applications the performance is diminished by the lack of locality.

2.3 Sparse Matrix Format

A number of memory efficient sparse matrix storage formats have been proposed to reduce the footprint. Here we give a brief overview of some of the most common formats. Many derivations (blocking, sawtooth, etc.) are based around one of the following formats. We use the matrix in Figure 1 to show how each format stores it data.

2.3.1 *Coordinate Format.* The coordinate format (COO) is the most intuitive approach to storing a sparse matrix: for each non-zero element it stores its row position, column position, and value. As shown in Figure 2(a) each data point is stored in a separate array. Each non-zero element is assigned a unique index, and this index is used to access each array. In our example we assign the indices in sequential order, but this is not a requirement of COO. Non-zero elements can be spread randomly throughout the array so long as the row, column, and value share the same index within their respective arrays.

$$\begin{bmatrix} 2 & 0 & 7 & 6 & 5 \\ 0 & 8 & 1 & 0 & 1 \\ 4 & 9 & 0 & 0 & 5 \\ 0 & 0 & 7 & 6 & 0 \end{bmatrix}$$

Fig. 1. An example matrix that is used to show how different sparse matrix formats store data.

2.3.2 *Compressed Sparse Row.* The compressed sparse row (CSR) is a natural extension to the COO format and is the most commonly used format. It stores all the column position, and value points in two separate arrays; just like COO. However, it saves memory by compressing the row position. As shown in Figure 2(b) some row positions are repeated multiple times. Instead of storing repetitive values the CSR format uses a row_ptr array, which only points to the first element of each row. To be functionally correct the column, and value array must be sorted by row such that all elements in row i are placed before any element in row $i + 1$.

The compressed sparse column format (CSC) is almost identical to CSR, but instead of compressing the row array it compresses the column array. The CSC format is less widely used because its workloads are harder to distribute evenly. SpMV has one output per row (not per column), and with CSR each output can be computed within a single processing element (PE). CSC would require synchronization across multiple PEs to prevent race conditions.

2.3.3 *ELLPACK.* The ELLPACK (ELL) format rose in popularity with vector processors. Peak performance required each processor to run the same workload. To achieve this ELL would zero pad the smaller rows until they were the same length as the largest. Figure 2(c) shows how our example matrix would be stored in ELL format. Because all rows are the same length the ELL only requires 2 arrays; one for the columns, and one for the values. The column position for zero padded data can have any value, but it should be less than the number of columns in the matrix to avoid out-of-bound memory accesses.

ELL is not as general a sparse matrix format as COO or CSR. It is only useful when all rows within a matrix have a small variance on the number of non-zero elements per row. If one, or a few rows are much larger than the rest it can cause too much zero padding. The extra work from processing zero values would eliminate any of performance gains.

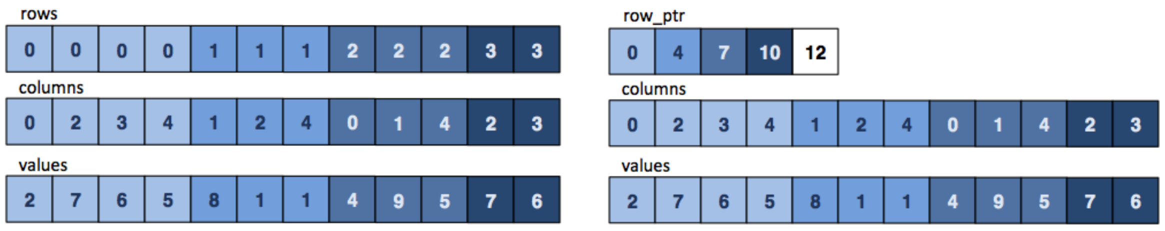
2.3.4 *Hybrid format.* A hybrid (HYB) format was proposed by [11] for GPU accelerators. It splits the matrix storage between the ELL, and the COO formats. Data that can be reasonably (low zero padding) stored in ELL will be, and the remaining values are stored in COO format. This method both reduces the zero padding caused by longer rows, and it still allows the GPU to benefit from ELL's structured memory; at least for part of the computation.

3. RELATED WORKS

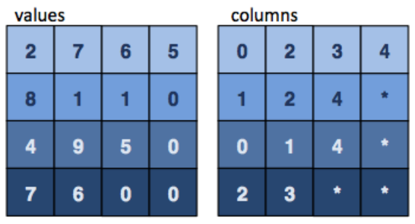
3.1 Multi-threaded Architectures

Multi-threading, as a latency masking approach, was initially proposed for the Denelcore HEP [31] and subsequently developed for the Horizon architecture [28]. The Horizon used 256 custom processors connected to global memory. Memory accesses took on average 50 to 80 clock cycles, but most all of the requests were fulfilled within 128 cycles. With this knowledge the custom processors were built to support 128 concurrent threads, and it could context switch between threads in one cycle. Therefore the memory latency could be masked, and the processor was always utilized by active and waiting threads. In the 1990s the Tera Corporation, later bought by Cray, developed such a design as a commercial machine. The Tera MTA [9; 8; 32] again had 256 processors all sharing 64 GB of memory organized as a distributed NUMA architecture connected to the processors via a sparsely populated regular topology interconnection network. To lower network traffic it forced instruction requests through a shared cache. Processors ran at 220 MHz, and could issue one memory request per cycle. A later design, the Cray XMT [19] machine, increased the number of supported processing cores to 8192, and the clock frequency to 500 MHz.

The Sun Microsystem UltraSPARC T1 and T2 microprocessors were designed, from scratch, as multi-threaded processors with the objective of achieving high throughput [14; 27; 13]. The execution is switched,



(a) COO format (b) CSR format



(c) ELL format

Fig. 2. COO, CSR and ELL sparse matrix storage formats for the exampole in Figure 1.

every cycle, to another ready thread. Upon a long latency operation, such as a cache miss, the thread is removed from the ready list.

3.2 SpMV Acceleration in Hardware

The MT-FPGA design was first proposed in [24] and its performance compared to multi-core CPU architectures reported in [36]. Our findings showed MT-FPGA outperformed AMD and Intel processors, but the STI-Cell processor was able to sustain higher overall throughput. However, the MT-FPGA showed a much higher utilization (FLOPS/cycle). We extend this work by comparing the MT-FPGA approach to FPGA and GPU hardware accelerators.

In [29] the authors proposed a cache based FPGA design for a Convey HC-1. We compare its performance against MT-FPGA on a Convey HC-2ex. Caching better utilizes the memory channels, and allows them to instantiate 32 PEs while the MT-FPGA instantiates only 20 PEs. However, the empirical results shown in this paper reveal a higher sustained throughput for the MT-FPGA.

Many researchers have focused on building efficient floating-point multiply accumulate (MAc) circuits on FPGAs for SpMV. The approaches in [33; 39] combine multiple adder IP cores into a tree structure with a feedback loop. The approach in [18; 30] statically assigns partial dot-products to multiple PEs. In [37] multiple MAcs are placed on a single FPGA, which allows the processing of multiple rows in parallel. Finally, [22] builds a single reduction circuit that can manage multiple rows through a single floating-point adder.

The previous approaches rely on the on-chip storage of the vector and matrix which limits the size and scope of applications. The approaches in both [29] and [24] rely on the off-chip storage of the vector and the matrix. Doing so removes all irregularity and allows the SpMV to be computed in a streaming fashion.

The CUSP-library [7] was developed at Nvidia for sparse linear algebra and graph computations. It supports multiple sparse matrix formats including all those highlighted in this paper. The librarie's performance benefits over multi-core CPU architectures were shown in [11].

4. THE CHAT COMPILER

CHAT is a C to VHDL compiler that targets irregular applications. It generates custom multi-threaded data-paths based on high level descriptions given by developers. In this section we provide a taxonomy for irregular applications, and explain how CHAT can be used to compile such applications.

4.1 A Taxonomy of Irregular Applications

Different types of irregular applications will require different hardware components, or will require the hardware components to be connected differently. CHAT can help developers quickly generate hardware applications that fit within its supported design paradigms. We propose four classes of irregular applications

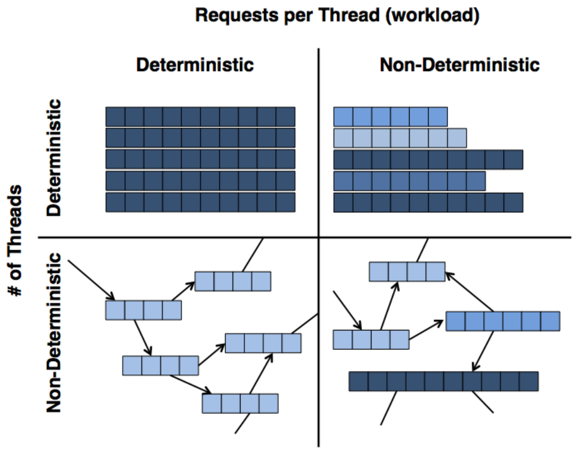


Fig. 3. Irregular applications can be classified based on whether or not the number of threads and the workload size are deterministic.

as shown in Figure 3. Classes are based on whether or not the number of threads, and the number of memory requests per thread (workload) are determinable.

Applications with a deterministic number of threads are statically analyzed and compiled to reduce or eliminate redundant execution. However, applications with a non-deterministic number of threads need extra FPGA constructs to prevent redundancy. Consider breadth first search; during graph traversal two nodes, A and B, could both point to the same node C. Without synchronization two threads would be generated to processes node C; causing repetitive and unnecessary computations. Performance would decrease exponentially as each thread then generates more and more redundant threads.

Applications with a deterministic workload, static number of memory requests per thread, can be assigned to processing engines (PEs) in round robin fashion. It is reasonable to expect threads that starting which start at the same time will finish at the same time when they perform the same number of operations. However, applications with non-deterministic workloads can have variable memory requests, and therefore a variable number of operations to perform. An unreasonable amount of stalling could occur with round-robin assignment because of unbalanced threads. A Thread Management Unit (TMU), local to the FPGA, can ensure a balanced execution by dynamically assigning threads as PEs become available. In this paper we explore the compilation of custom TMUs for FPGA kernels.

4.2 C to HDL Tools

Many high level synthesis (HLS) tool have been proposed before CHAT. Their aim is to compile a high level language (HLL), like C/C++ or Java, into a hardware description language (HDL), like Verilog, VHDL, or System-C. Using a compiler to bridge the gap between HLLs, and HDLs brings hardware accelerators from the domain of hardware specialists to the domain of traditionally trained application developers. Impulse C [2], ROCCC [3], and Vivado-HLS (originally AutoPilot) [38; 6] each work with a subset of the C language to create custom IP accelerators. These tools do not specify a system end-to-end, but assist in creating the individual components for a RTL design. Some languages will support more HLL constructs than others. For instance, Catapult C [1] supports pointers, classes, templates when generating custom cores. Other HLS tools, like Bluespec [5], do not use established languages. Instead they use their own HLLs, like SystemVerilog, to describe hardware at a level above typical HDL languages.

ROCCC [35] is a C-to-VHDL compiler which targets streaming (regular) applications so that it can analyze them and generate an optimized data-path. It attempts to minimize a kernel's outgoing memory requests, minimize its area, and maximize its clock frequency. Developers write the kernel in C and the compiler applies its optimizations at two levels each with its own intermediate representation (IR); Hi-CIRRF and Lo-CIRRF [23]. Hi-CIRRF identifies FPGA components, which will be needed during execution. It marks and inserts these into the IR as C macros. Hi-CIRRF is also responsible for replicating any kernel's logic, depending on user specifications, for parallel execution; such as loop unrolling. Lo-CIRRF generates a data flow graph (DFG) as another IR, which eventually becomes the kernel's data path. Control systems are built

```
for (a = 0; a < a_size; ++a) {
  for (b = 0; b < b_size; ++b) {
    out[a][b] = … ;
  }
}
```

(a) Thread with a static workload

```
for (a = 0; a < a_size; ++a) {
  tmp = 0;
  for (b = C[a]; b < D[a]; ++b) {
    tmp = … ;
  }
  out[a] = tmp;
}
```

(b) Thread with a dynamic workload

Fig. 4. Compilable FPGA kernels with static and dynamic workloads. The highlighted region shows a threads workload.

around the DFG to manage execution flow and memory requests.

The CHAT [26; 25] compiler uses the same underlying tools as ROCCC for Hi-CIRRF and Lo-CIRRF compilation, but it targets irregular applications. Initially, it focused on irregular kernels with a deterministic number of threads, and each thread had a deterministic workload. In this work we extend its capabilities to kernels with non-deterministic workloads.

4.3 CHAT Compiler Syntax

In CHAT a for-loop supports a subset of its traditional C functionality. The loop has a very strict purpose, which is to define how data streams (arrays) will be accessed. CHAT only supports one loop index per for-loop declaration. However, once declared these indices can be read freely in the loop body. Nested for-loops are supported for more complex kernels.

The structure for kennels with static workloads is shown in Figure 4(a). Multiple variables can be used to route logic through the data path, but all logic must be defined within the innermost for-loop's body. As shown, variables a and b are indices to the *out* data stream, but they could also be used to index other streams. Both a and b must start at a static position, but they can go to a variable position which must be set before the execution begins. Because this variable cannot change, all threads will have the same workload though they may have different data.

The new compiler semantics support kernel constructs for dynamic workloads. Kernel logic can be defined outside the innermost for-loop, and it removes restrictions on a inner for-loop's initialization and condition sections as shown in Figure 4(b). Threads can support dynamic workloads because a loop's start and end conditions can change during execution. A thread's body is defined by the lowest for-loop with static start and end conditions at runtime. In Figure 4(b) a threads workload is defined by the outermost for-loop's body.

4.4 CHAT Data Path Construction

To support threads with dynamic workloads the compiler must generate additional FPGA constructs. For this purpose a Thread Management Unit (TMU) is introduced, and the data path is broken into two parts; thread management, and processing. Both are custom generated by the compiler for each kernel. The TMU maintains threads needing to be processed and dispatches them to aPE. A PE can handle multiple threads in parallel, which ensures enough request are generated to mask long memory latencies. A PE manages the states for all threads that are assigned to it. Once a thread completes the PE sends its output back to the TMU. The TMU then sends the output on to global memory and removes the thread.

Parallelism can be improved by increasing the number of generated PEs, which is tunable to individual architectures by way of compiler arguments. While the number of PEs can be increased there is one TMU generated. The TMU creates the start and end conditions for a thread and dynamically assigns it to an

```
void spmv_csr (int *row, int *val, int *col,
               int *vec, int *out, int length) {
  int r, c, tmp;

  for (r = 0; r < length; ++r) {
      for (c = row[r]; c < row[r+1]; ++c ) {
         tmp = tmp + val[c] * vec[ col[c] ];
      }
      out[r] = tmp;
   }
}
```

Fig. 5. The source code used to generate a custom multithreaded SpMV kernel.

available PE. This maintains a balanced execution. A Thread with a disproportionately large workload will occupy one PE while the other PEs handle multiple threads. Because the workload can be unbalanced the kernel does not guarantee in-order thread completion. Each PEs hold a busy flag high to prevent additional threads from being assigned. When all PEs are busy the TMU back loads threads to quickly assign them later.

Identification of kernels with dynamic workloads is done during the compiler's Hi-CIRRF pass. The compiler uses a for-loop's initialization, and condition section to make this determination. Consider the inner loop from Figure 4(b). It is initialized with values from stream C, and its condition is dependent upon stream D. When this happens the compiler inserts a TMU macro into the IR and links it to streams C and D. Lo-CIRRF takes the macro and build a custom TMU for the kernel.

5. MULTI-THREADED SPMV EXAMPLE

We have used the CHAT compiler tool to generate a custom multithreaded FPGA Sparse Matrix Vector multiplication (SpMV) kernel targeting a Convey HC-2ex machine. In this section we outline the sparse matrix data structure used, one method a kernel developer could use to write the FPGA's SpMV code, how the code is compiled to an FPGA circuit, and how each of the FPGA components interact. This section also outlines how the TMU creates and manages threads. This section concludes with how the kernel is integrated into the HC-2ex.

5.1 SpMV Kernel Code

Sample code for a SpMV kernel is shown in Figure 5. This is, line for line, the code used by the compiler. All arrays are treated as streams of data into the FPGA. Most (*row*, *val*, *col*) are accessed in a streaming (regular) fashion. The *vec* array is accessed by the *col* array and as such is treated as an irregular accesses. Thread workloads are determined by the *row* stream with the two adjacent elements giving start and end positions. Threads are issued in order, but they are not required to have the same workload size. Thus the *out* array can write to memory out-of-order. The kernel writes whenever a thread completes. The designer can unroll the outer for-loop to generate multiple PEs yielding higher parallelism and throughput.

5.2 Processing Element

The bulk of the SpMV's work is done by the processing element (PEs). These engines operate independently and the number of PEs is only limited by the resources available to the FPGA. Each thread assigned to a PE will generate one output, which is the sum-of-products for the row. One thread is generated for each matrix row, and it holds the start and end position for the memory requests. As requests are fulfilled the data is sent to a summation unit which produces the final sum-of-products. One PE can manage the requesting, multiplying, and summing of multiple threads (rows) concurrently.

The main components of a PE are shown in Figure 6. Each PE must manages the memory requests to the column, value, and vector arrays. The Convey architecture supports the in-order return of all memory requests. The reordering is done by a crossbar that interfaces the HC-2ex's FPGAs to the memory modules. Because memory is returned in-order the kernel can use FIFO buffers to store data.

When assigned a new thread the PE will raise a busy flag and incrementally request memory locations from the column and value arrays. When all the requests for that thread have been issued, the flag is lowered and a new thread is assigned to the PE. This is done even though all the data of the prior thread has not

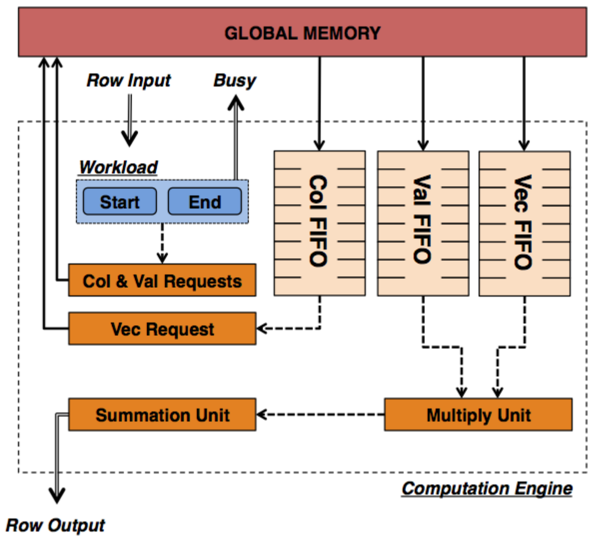


Fig. 6. Each PE is assigned a thread. It requests the necessary data (Column, Vector, and Value) from global memory. Returned data values are pushed through the multiply pipeline, and summation unit.

yet been returned. Workloads are balanced across PEs because they do not accept new threads until all requests for previous threads have been issued. FIFO buffers within each PE are large enough to support the outstanding memory requests. As memory returns the data for the column array is used to generate the memory requests for the vector array. The data returned for the value array is held in buffer until the corresponding vector request is fulfilled.

As data is returned from memory it is buffered in the Value and Vector FIFOs with its thread id. For this kernel a thread id is the row's index. The summation unit uses the thread ids to manage concurrent threads. SpMV reductions circuits have been widely studied. Our compiler uses a circuit similar to the one described in [22]. It handles multiple rows concurrently, and can read a new element every cycle. However, the circuit assumes data for one row is sent, in its entirety, before another row begins. This assumption holds for the kernel and Convey HC-2ex architecture. This reduction circuit is only needed if the kernel is compiled for floating point operations, which require multiple cycles for each multiplication-addition.

5.3 Thread Management

The Convey HC-2ex has 4 Virtex 6 LX760 FPGAs, called application engines (AEs). Figure 8 shows the SpMV kernel layout for one HC-2ex AE. The design can be replicated to all four AEs at runtime. Each AE has 16 memory channels, and each PE requires 3 memory channels; for the column, value, and vector requests. Memory channels are this design's bottleneck, which is limited to 5 PEs per AE.

Control registers in the AE specify the number of threads (rows) needed by the SpMV kernel. The registers also specify the base addresses for all streams used by the kernel. These values can be assigned unique values when multiple AEs are used. Each AE can processes a subset of the overall matrix.

One thread management unit (TMU) communicates with the 5 PEs. It creates thread workloads with value pairss from the row pointer array. Access to the row pointer incurs the same memory latency as all other requests. The TMU buffers threads when all PEs are busy. Assignment is done dynamically as PEs become available.

As PE threads complete the output value is buffered by the TMU until it can be written back to global memory. The TMU manages 5 *out* streams (one per PE) as well as the *row* stream. Reads and writes to these two streams are infrequent, occurring once per thread, compared to the column, value, and vector streams. The TMU uses one memory channel, and interleaves its read and write requests as shown in Figure 7. Read and write request conflicts are resolved by a control unit in favor of the write data, which prevents a deadlock.

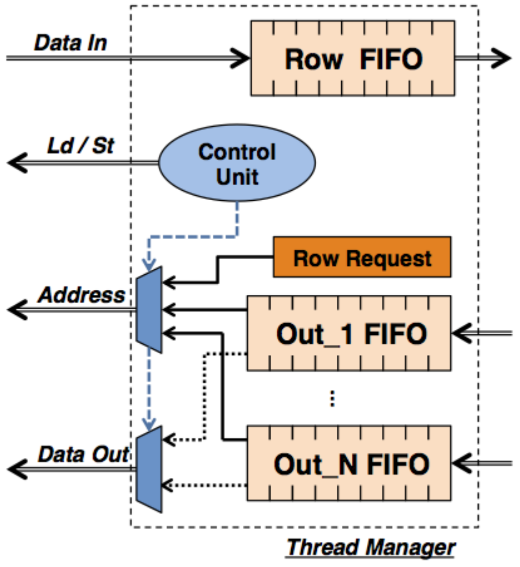


Fig. 7. Thread Management Unit: the output write data and row requests are combined into one memory channel. A control unit handles the conflicts when multiple resources need to use the channel.

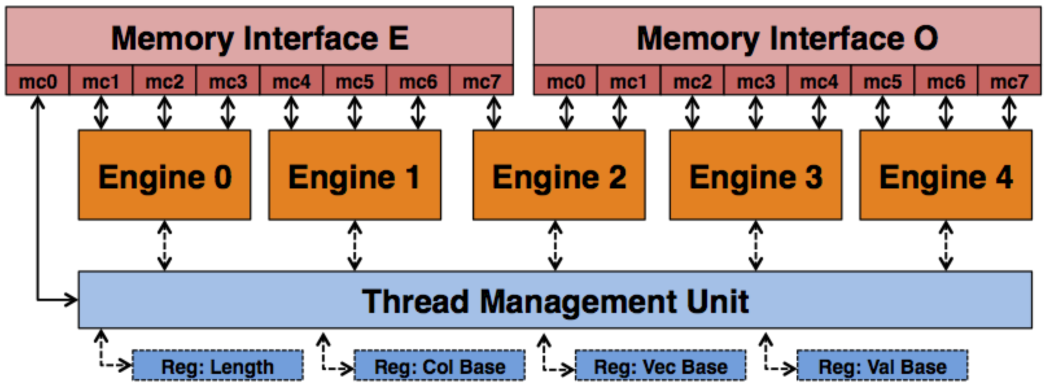


Fig. 8. The MT-FPGA architecture on one AE. Control signals specify the number of jobs (length), and the base addresses of the sparse matrix arrays. All memory channels of the AE are utilized.

5.4 Convey HC-2ex Implementation

Integrating a kernel into the HC-2ex does not require that all memory channels be utilized, but with SpMV more channels used implies more PEs used. As discussed above each PE requires 3 channels, and the TMU interleaves its reads and writes through a single channel. Therefore, we can use use between 1 (4 memory channels) and 5 (all 16 memory channels) PEs on a single AE. In Table 1 we show how the FPGA slices, and BRAMs increase as we scale the design from 1 to 5 PEs. This gives us insight into how well the design could scale on another platform with more, or less memory channels. These results include Convey's memory interface wrapper, which is not a significant portion of the area. We see that even with 5 PEs the Virtex-6 still has plenty of room for more engines; only 33% of the logic is utilized. The current design is therefore limited by the memory channels.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

Table 2 shows the architecture specifications for the hardware accelerators used in this paper. The reader should note that [29] used a Convey HC-1 FPGA which is different from the Convey HC-2ex used for our

Table 1. FPGA resource utilization when varying the number of PEs per FPGA.

PE(s)	Slices (118,560)	BRAMs (720)
1	25,788 (21%)	107 (14%)
2	29,040 (24%)	133 (18%)
3	32,638 (27%)	179 (24%)
4	36,520 (30%)	209 (29%)
5	39,395 (33%)	239 (33%)

Table 2. Architecture specifications for the various hardware accelerators used in this paper.

Company	Platform	Type	Clock (MHz)	Cores	Memory Size	Memory Bandwidth
Convey	HC-2ex	FPGA	150	n/a	128 GB	76.8 GB/s
Nvidia	GTX 280	GPU	1296	240	1 GB	141.7 GB/s
Nvidia	Tesla C1060	GPU	1300	240	4 GB	102 GB/s
Nvidia	Tesla K20	GPU	705	2496	5 GB	208 GB/s

MT-FPGA approach. As previously stated, the HC-1 has 4 Xilinx Virtex-5 FPGAs, and the HC-2ex has 4 Virtex-6 FPGAs. Regardless, both memories have a peak memory bandwidth of 76.8 GB/s and a memory clock rate of 150 MHz. The Virtex-6 does have more area, but neither design is area bounded so this is of little concern. The GPU results obtained by [11] were run on a Nvidia GTX 280, which was a large GPU for the time (2008). However, since then Nvidia's GPU architecture has evolved from Fermi to Kepler. In this paper we report new GPU results on a Tesla K20 (Kepler architecture). We also replicate the old GPX results on a similar Tesla C1060 GPU. All GPU results were obtained using the CUSP-library [7]. Results on the Tesla K20 used CUDA version 6.0, and GCC version 4.6.3. Results on the Tesla C1060 used CUDA version 4.1, and GCC version 4.1.2. Throughput results, for all the tests we ran, are reported in Table 6.

The benchmarks evaluated in this paper, also shown in Table 6, were obtained from the University of Florida Sparse Matrix Collection [4]. Most of them were chosen to directly compare with previously reported results. We break the benchmarks into three distinct suites for clarity.

Suite 1 consists of the first six benchmarks (dw8192 to torso2). These were used in [29].

Suite 2 is the next 14 benchmarks (Dense to LP) were used in [36] and [11]. The names, for some of these 14 benchmarks, are changed from their UF collection names. We did this to match how they were presented in [36; 11].

Suite 3 is the last four bencharks (cage_15 to Serena). These wre included to evaluate the performance of the accelerators on very large benchmarks that are more indicative of some of real scientific, economic and engineering workloads.

6.2 Memory Footprint of Sparse Matrix Formats

Throughput on the MT-FPGA design is contingent only on how much data has to be streamed to the FPGA. This differs from modern CPU and GPU architectures, which require effective caching to maintain a high throughput. Memory masking should be effective regardless of the latency (assuming a large enough dataset), or any number of cache misses. Therefore, the MT-FPGA kernel's priority is to use a sparse matrix format with the smallest memory footprint.

In Table 3 we compare the memory footprint of common sparse matrix formats. Intuitively we know the CSR footprint will always be smaller than that of COO; except in the special case where there is only one element per row. They have the same three arrays, but CSR does compression on the row. ELL, and HYB are less clear. ELL uses two matrices, and assumes the rows size to be static. If the sparse matrix is uniform, then this format would require less memory than CSR. HYB is a combination of ELL and COO, which makes its footprint difficult to estimate as well. However, the empirical results from Table 3 show that CSR usually has the smallest footprint.

6.3 Dense Matrix Experiments

The MT-FPGA approach is designed to cope with long memory latencies by issuing many outstanding requests. However, performance is dependent upon the dataset, and the architecture's memory latency. Early stages of execution will be dominated by memory request until the kernel's TMU can buffer extra

Suite 1

Benchmark Name	Rows	Non-Zero	NNZ/Rows	MT-FPGA	Tesla C1060	Tesla K20
dw8192	8,192	41,746	5	1.88	1.79	4.79
t2d_q9	9,801	87,025	9	2.74	1.54	5.95
epb1	14,734	95,053	6	2.51	1.62	5.51
raefsky1	3,242	294,276	91	3.66	1.84	2.89
psmigr_2	3,140	540,022	172	2.76	0.38	0.72
torso2	115,967	1033,473	9	3.48	1.59	6.63
Geo Mean				2.77	1.03	3.61

Suite 2

Benchmark Name	Rows	Non-Zero	NNZ/Rows	MT-FPGA	Tesla C1060	Tesla K20
Dense	2,000	4,000,000	2,000	4.39	0.6	2.24
Protein	36,417	4,344,765	119	3.81	1.33	1.81
FEM/Spheres	83,334	6,010,480	72	4.32	1.54	1.60
FEM/Cantilever	62,451	4,007,383	64	4.25	1.48	1.74
Wind Tunnel	217,918	11,634,424	53	4.14	1.67	1.64
FEM/Harbor	46,835	2,374,001	51	3.85	1.33	2.21
QCD	49,152	1,916,928	39	3.98	1.66	1.81
FEM/Ship	140,874	7,813,404	55	4.15	1.51	2.63
Economics	206,500	1,273,389	6	2.89	1.29	4.68
Epidemiology	525,825	2,100,225	4	2.04	2.58	9.63
FEM/Accelerator	121,192	2,624,331	22	1.91	1.32	2.31
Circuit	170,998	958,936	6	2.53	1.23	4.04
Webbase	1,000,005	3,105,536	3	0.94	0.78	1.68
LP	4,284	11,279,748	2,633	4.49	0.23	0.54
Geo Mean				3.16	1.18	2.23

Suite 3

Benchmark Name	Rows	Non-Zero	NNZ/Rows	MT-FPGA	Tesla C1060	Tesla K20
cage15	5,154,859	99,199,551	19	4.05	1.54	2.32
circuit5M	5,558,326	59,524,291	11	1.24	0.09	0.19
Flan_1565	1,564,794	117,406,044	75	4.22	1.54	1.57
Serena	1,391,349	64,531,701	46	4.16	1.35	1.61
Geo Mean				3.06	0.73	1.03

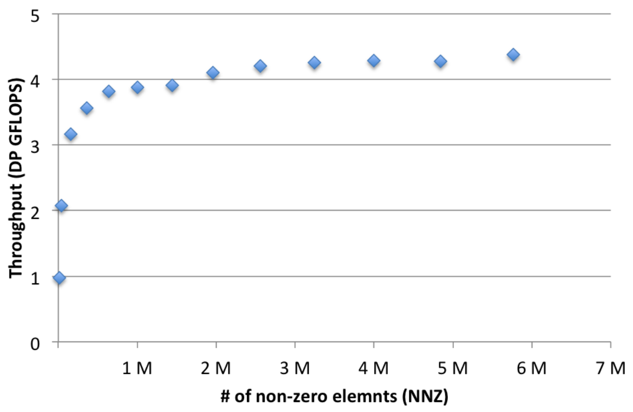


Fig. 9. Throughput of 20 PEs running on different dense matrices stored in CSR format.

threads. This cost will negatively effect smaller matrices. We run a series of test on the Convey HC-2ex to determine what size datasets are "large enough" to mask the initial startup costs.

This experiment uses dense matrices stored in CSR format. In doing so we remove all irregularity, and therefore get a better estimate of the startup cost. With no irregularity any optimizations or caching implemented Convey's memory crossbar will yield their best performance, and in turn give us the minimum dataset size needed to mask the startup costs. The tests fully utilize the HC-2ex's memory channels; 20

Table 3. The memory storage requirement (MB) for each benchmark in the three sparse matrix formats. ELL pads all smaller rows to the length of the longest. When this is unreasonably long (> 10 GB) we mark its size as N/A.

Suite 1

Benchmark Name	CSR	ELL	HYB
dw8192	0.7	1.0	0.7
t2d_q9	1.5	1.4	1.4
epb1	1.6	1.7	1.7
raefsky1	4.7	5.6	7.1
psmigr_2	8.7	115.0	13.0
torso2	17.5	18.6	16.7
AVERAGE	5.8	23.9	6.7

Suite 2

Dense	64.0	64.0	96.0
Protein	69.8	118.0	84.6
FEM/Spheres	96.8	108.0	108.0
FEM/Cantilever	64.6	77.9	75.3
Wind Tunnel	187.0	631.0	189.0
FEM/Harbor	38.4	108.0	51.9
QCD	31.1	30.7	30.7
FEM/Ship	126.0	229.0	134.0
Economics	22.0	145.0	28.9
Epidemiology	37.8	33.7	33.7
FEM/Accelerator	43.0	46.5	55.5
Circuit	16.7	965.0	18.7
Webbase	57.7	N/A	58.7
LP	180.0	3,850.0	270.0
AVERAGE	74.0	196.0	88.3

Suite 3

cage15	1,620.0	3,870.0	1,920.0
circuit5M	996.0	N/A	1,200.0
Flan_1565	1,890.0	N/A	2,020.0
Serena	1,040.0	4,470.0	1,190.0
AVERAGE	1,380.0	N/A	1,580.0

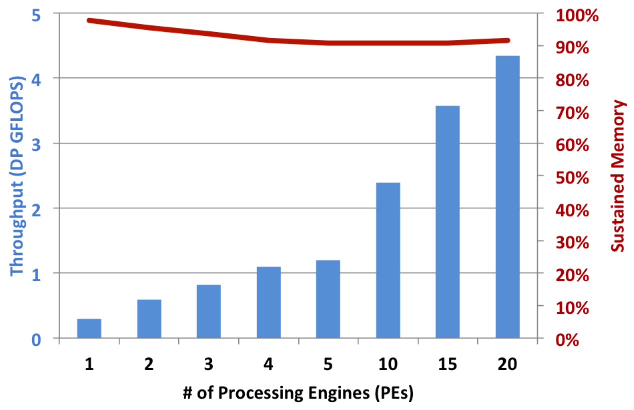


Fig. 10. Throughput and sustained memory bandwidth of varying PEs on a 4 million element dense matrix stored in CSR format.

PEs across all four AEs. The results are shown in Figure 9. Sustained throughput quickly approaches 4 GFLOPS as the number of non-zero elements (NZE) approaches 1-million. From there it slowly rises to a peak throughput of 4.5 GFLOPS. The design sustains 75% of the HC-2ex's peak throughput, and based on these results we conclude 1-million NZEs as the minimum dataset size needed to mask the startup costs. An

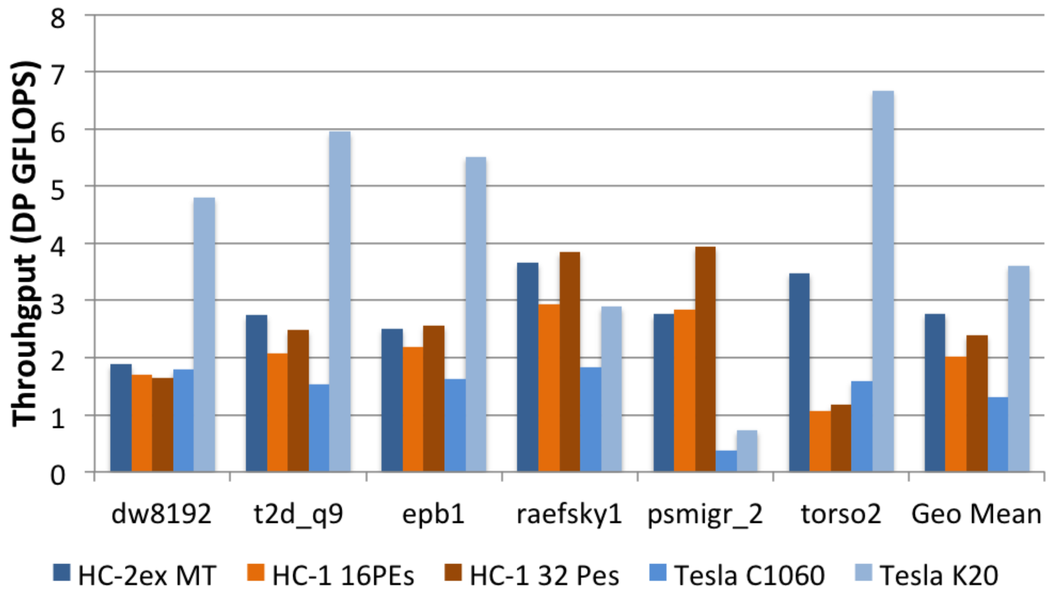


Fig. 11. FPGA and GPU performance results vs HC-1 [29] results on benchmark Suite 1.

architecture with longer or shorter memory latencies will likely need a larger or smaller dataset.

We were also interested to know how well the memory system would cope as the number of utilized channels increased. Again we used a dense matrix stored in CSR format. All tests use a 4 million element (2,000 x 2,000) matrix. Starting with a single AE we increased the number of engines from 1 (3 channels) up to 5 (15 channels). Then we increase the number of AEs used from 1 (15 channels) to 4 (60 channels). Results are shown in Figure 10. The bar chart reports the sustained GFLOPS, and the red line reports the percentage of peak memory bandwidth sustained. The efficiency with a small number of engines is very close to optimal, which suggests the number of requests is not high enough to saturate the memory. However, it slowly drops to 90% as one AE is filled with five PEs. The design scales linearly from one (5 PEs) to all four AEs (15 PEs). These results shows that MT-FPGA will scale well at 90% efficiency so long as the memory system can handle the requests.

6.4 Throughput Comparison using CSR

We evaluate the throughput (DP FLOPS) performance of our MT-FPGA approach, and our GPU results against previously published results. Suite 1 was used in [29] and consists of smaller matrices. The largest matrix in this set is 1 million NZEs, but the average size is only 350,000 NZEs. Suite 2 was used in [11] and consists of medium-sized matrices. The size ranges from 1 to 11 million NZEs with an average of 4.5 million. We introduce Suite 3 to evaluate the performance on large matrices. These range from 60 to 117 million NZE. It should be noted before these comparisons that all GPUs have both a higher clock frequency than the FPGA, and a larger memory bandwidth.

Figure 11 shows the results for Suite 1. HC-2ex MT is the approach presented in this paper. Both HC-1 16 PE, and HC-1 32 PE are cached based approaches described in [29]. Both Tesla C1060 and K20 use the CUSP library. In these tests we did not expect the MT-FPGA to perform well because all matrices, except one, are below our empirical cutoff for being ”large enough”. Regardless the MT-FPGA design still outperforms the cache based FPGA designs with a mean throughput of 2.77 DP GFLOPS compared to 2.02, and 2.39 DP GFLOPS. This is achieved even though our MT-FPGA has 12 fewer PEs. However, this point is moot because the Kepler GPU dominates both of the FPGA designs. The Kepler architecture performed significantly better than the Fermi architecture. This shows how the GPU has progressed in the years since the initial studies.

Figure 12 shows the results for Suite 2. HC-2ex MT, Tesla C1060, and K20 are the same as before. GTX 280 is the GPU results presented in [11]. This dataset shows how consistent the MT-FPGA can be on certain

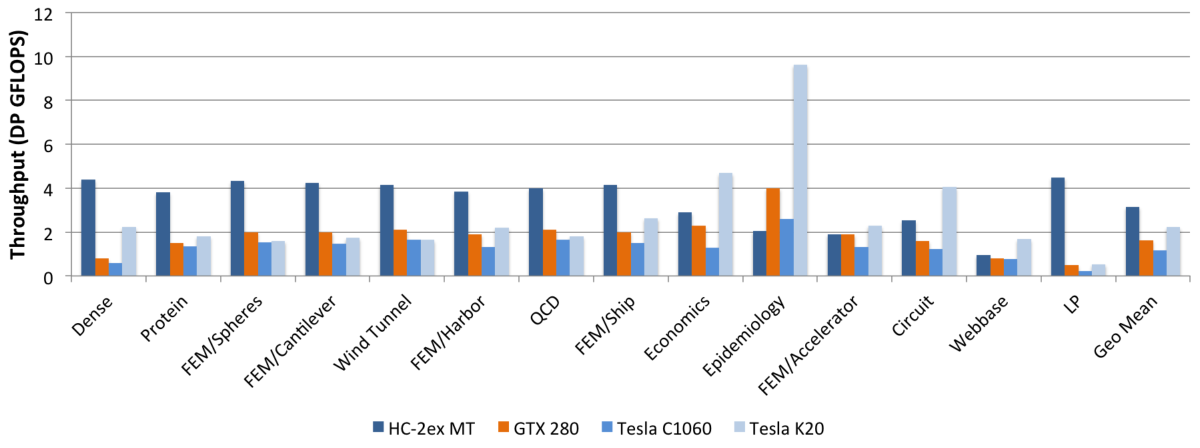


Fig. 12. FPGA and GPU performance results vs GTX 280 [11] results on benchmark Suite 2.

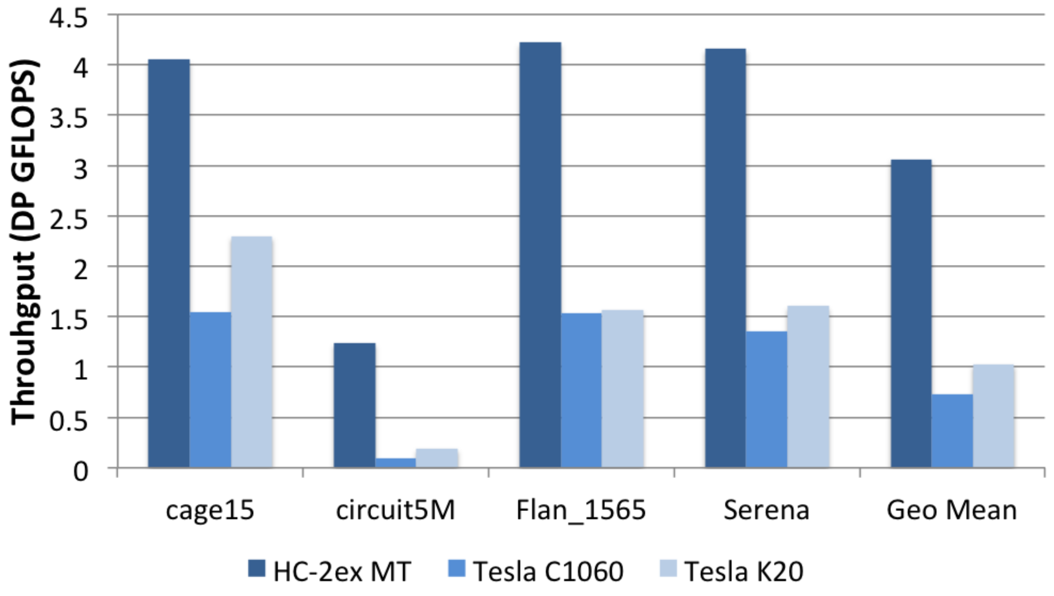


Fig. 13. FPGA and GPU performance results on benchmark Suite 3.

matrices. For many results the performance is around 4 DP GFLOPS. However, when the NNZ/row is small (< 10) the FPGA performance drops. We attribute this to the thread startup costs. When a PE is assigned a new thread it takes a few cycles to initialize its counters and ID managers. During this time a PE does not make any memory requests, and the effect is more pronounced when the row size is small. Regardless, the MT-FPGA outperforms all GPU approaches on average.

Figure 13 shows the results for Suite 3. Here we can see that the MT-FPGA performs substantially better than the GPUs. The FPGA has a mean performance of 3.06 DP GFLOPS while the Kepler GPU only achieved 1.03. Again the FPGA performance is consistent at 4 DP GFLOPS except on Circuit5M where the NNZ/row is 11.

6.5 GPU Throughput with Various SpMV Formats

In Section 6.4 we compared the FPGA and GPU throughput on the same sparse matrix format. We choose CSR because it is the format that made the most sense for MT-FPGA. However, this does not mean CSR is the best format for the GPU architecture. The GPU's SIMD paradigm rewards approaches that can reuse

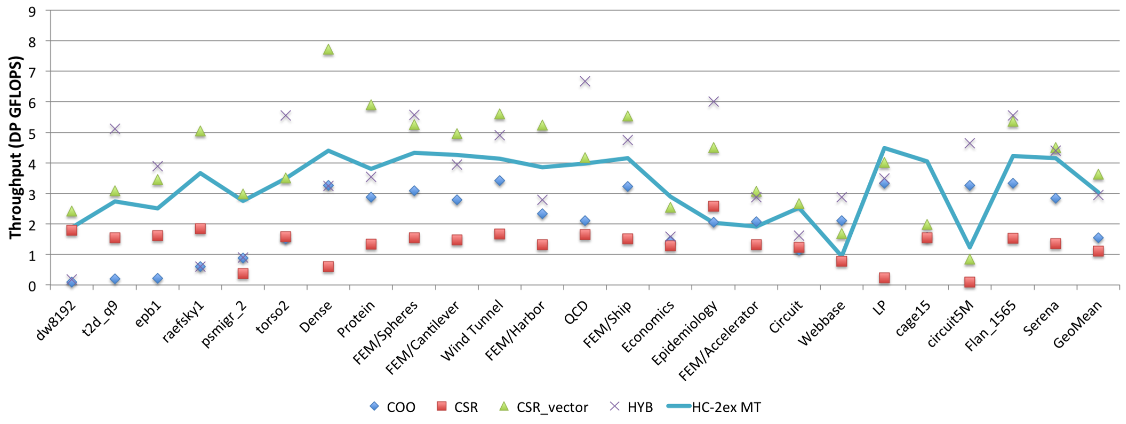


Fig. 14. Tesla C1060 GPU throughput performance on varying sparse matrix formats compared to the MT-FPGA's performance with CSR.

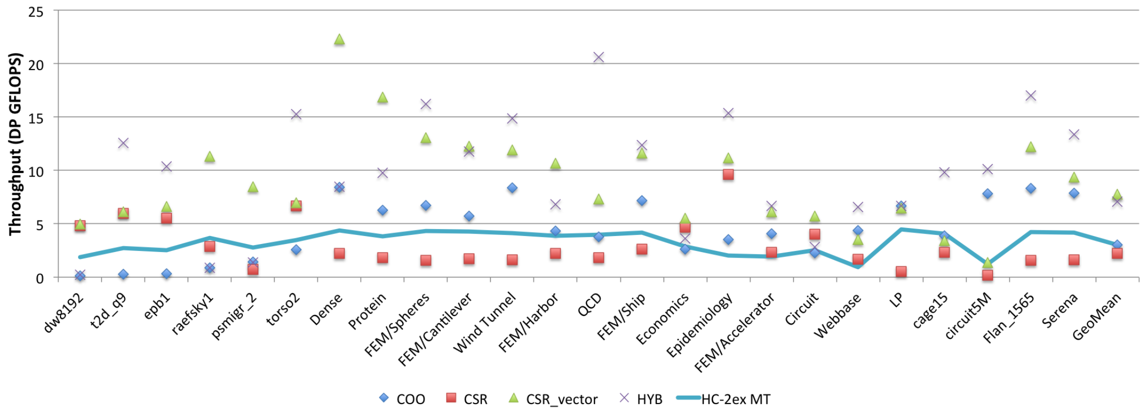


Fig. 15. Tesla K20 GPU throughput performance on varying sparse matrix formats compared to the MT-FPGA's performance with CSR.

instructions not necessarily approaches that stream the least data. Therefore, well structured formats like ELL, or HYB could offer better GPU performance.

Figure 14 and Figure 15 show the GPUs performance on different SpMV formats. Many ELL benchmarks were beyond the CUSP library's threshold for reasonable storage. Therefore we do not report results for ELL in this paper. In these figures we see that the hybrid format, and a vectorized CSR performs significantly better than the regular CSR format. Throughput, for the Kepler GPU is doubled compared to the MT-FPGA approach. However, these results can be misleading. While the current GPUs achieve double the performance, they also have almost 3x the memory bandwidth compared to MT-FPGA. As FPGA technology

Table 4. Throughput per memory bandwidth and processor utilization.

Architecture	SpMV Format	Memory Bandwidth Efficiency FLOP/KB	Processor Utilization FLOP/Cycle
MT-FPGA	CSR	39.6	0.543
Tesla C1060	COO	15.2	0.026
Tesla C1060	CSR_scalar	10.9	0.017
Tesla C1060	CSR_vector	35.6	0.051
Tesla C1060	HYB	28.9	0.047
Tesla K20	COO	14.6	0.004
Tesla K20	CSR_scalar	10.6	0.002
Tesla K20	CSR_vector	37.4	0.008
Tesla K20	HYB	34.1	0.008

advances so too should its performance.

In Table 4 we report the average performance (across all benchmarks) normalized to the available bandwidth (FLOPS/KB) and the processor utilization (FLOP/Cycle) for each accelerator architecture and sparse matrix storage format. The MT-FPGA performance using CSR (39.6 FLOP/KB) is higher than the best performing GPU matrix format the Tesla K20 with HYB at 34.1 FLOP/KB. The processor utilization of the MT-FPGA is one to two orders of magnitude larger than that of the GPU architectures. This shows that latency masking multithreading is an effective paradigm for achieving high-throughput computation even with relatively low clock speeds.

7. CONCLUSION

Rather than mitigate long memory latency with a multilevel cache hierarchy, multithreaded executions masks memory latency by switching to a ready thread upon a long latency memory access. Like pipelining, the concurrency achieved is equal to the number of in-flight memory operations. Multithreaded execution is particularly effective on irregular applications that exhibit very poor spatial and temporal localities.

In this paper we have described the MT-FPGA, an FPGA-based multithreaded execution where the data-path is customized by a compiler tool (CHAT) to each specific application. for It supports, in hardware, the switching to a waiting queue of threads waiting for a memory reply and the resumption of threads ready to execute. We describe the steps taken by CHAT to generate the customized data-path and the thread management unit for various types of irregular applications.

Using the SpMV application as example, we describe the implementation of MT-FPGA on the Convey Computers HC-2ex consisting of five processing engines (PEs) on each of the four accelerator FPGAs. The five PEs on each FPGA use one third of the available hardware resources but all of the memory bandwidth. We empirically show that one million non-zero elements (NZE) is the smallest matrix that makes a multithreaded execution cost effective.

Using three suites of benchmark sparse matrices we evaluate first the storage efficiency of three sparse matrix storage models and identify the CSR (compressed sparse row) as achieving the most compact storage. The throughput performance of the MT-FPGA on the HC-2ex is then compared to the Nvidia Tesla C1060 (Fermi) and K20 (Kepler) GPUs programmed using the CUSP library. Even though GPU architectures have a substantial clock frequency and memory bandwidth advantage the MT-FPGA outperforms both GPU platforms on large matrices using the CSR storage (3x higher throughput than the K20). However, the CSR storage model is not the best suited for SpMV on GPU architectures. We show that alternative storage models achieve much higher throughput on the K20 architecture. Nevertheless, the MT-FPGA does achieve the highest ratio of FLOP/KB of all the storage models and architectures evaluated. Furthermore, inspire of it having the lowest clock frequency and memory bandwidth, it achieves two orders of magnitude higher processor utilization (FLOP/cycle) than the high-end GPU architectures.

In summary, this paper validates the latency masking advantage of multithreaded execution and its ability to boost throughput even when limited by clock frequency and memory bandwidth.

Acknowledgments

This work was supported in part by NSF Awards CCF-1219180 and IIS-1161997

REFERENCES

[1] Catapult C. *http://www.mentor.com/*.

[2] Impulse C. *http://www.impulseaccelerated.com/*.

[3] Riverside Optimizing Compiler for Configurable Computing. *http://roccc.cs.ucr.edu/*, 2012.

[4] The University of Florida Sparse Matrix Collection. *http://www.cise.ufl.edu/research/sparse/matrices/*, 2012.

[5] Open System C Initiative. *http://www.accellera.org/home/*, 2013.

[6] Xilinx Vivado. *http://www.xilinx.com/tools/autoesl_instructions.htm*, 2013.

[7] CUSP-library. *http://cusplibrary.github.io/*, 2014.

[8] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *International Confonference on Supercomputing*, Supercomputing '92, pages 188–197, 1992.

[9] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, Supercomputing '90, pages 1–6, 1990.

[10] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *International Conference on Parallel Processing*, ICPP '06, pages 523 –530, 2006.

[11] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA, 2008.

[12] T. M. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, 30:70–79, March 2010.

[13] V. Cakarevic, P. Radojkovic, J. Verdu, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the ultrasparc t2 processor. In *IEEE/ACM international symposium on Microarchitecture*, MICRO '09, pages 481–492, 2009.

[14] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, 2005.

[15] Convey Computer. *Convey Computer Reference Manual*, 2009.

[16] Convey Computer. *Convey Computer Programmer's Guide*, 2010.

[17] Convey Computer. *Convey Computer PDK Reference Manual*, 2012.

[18] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 75–85, New York, NY, USA, 2005. ACM.

[19] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[20] E. Fernandez, W. A. Najjar, and S. Lonardi. String matching in hardware using the fm-index. In *FCCM*, pages 218–225, 2011.

[21] E. Fernandez, W. A. Najjar, S. Lonardi, and J. R. Villarreal. Multithreaded fpga acceleration of dna sequence mapping. In *HPEC*, pages 1–6, 2012.

[22] M. Gerards, J. Kuper, A. Kokkeler, and B. Molenkamp. Streaming reduction circuit. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 287 –292, August 2009.

[23] Z. Guo and W. Najjar. A compiler intermediate representation for reconfigurable fabrics. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4, 2006.

[24] R. J. Halstead and W. Najjar. Compiled multithreaded data paths on fpgas for dynamic workloads. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 1–10, 2013.

[25] R. J. Halstead and W. Najjar. Compiling irregular applciations for reconfigurable systems. In *International Jounal on High Performance Computing and Networking*, 2013.

[26] R. J. Halstead, J. Villarreal, and W. Najjar. Exploring irregular memory accesses on fpgas. In *Proceedings of the workshop on Irregular applications: architectures and algorithm*, IAAA '11, pages 31–34, 2011.

[27] G. Konstadinidis, M. Rashid, P. F. Lai, Y. Otaguro, Y. Orginos, S. Parampalli, M. Steigerwald, S. Gundala, R. Pyapali, L. Rarick, I. Elkin, Y. Ge, and I. Parulkar. Implementation of a third-generation 16-core 32-thread chip-multithreading sparc processor. In *IEEE International Conference on Solid-State Circuits*, ISSCC '08, pages 84–597, 2008.

[28] J. T. Kuehnand and B. J. Smith. The horizon supercomputing system: Architecture and software. In *IEEE Conference on Supercomputing*, Supercomputing '88, pages 28–34, 1988.

[29] K. Nagar and J. Bakos. A sparse matrix personality for the Convey HC-1. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Ann. Int. Symp. on*, pages 1 –8, may 2011.

[30] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. FPGA and GPU implementation of large scale SpMV. In *Application Specific Processors (SASP), 2010 IEEE 8th Symp. on*, pages 64 –70, June 2010.

[31] B. J. Smith. Architecture and applications of the hep multiprocessor computer system. In *International Society for Optics and Photonics*, SPIE'82, pages 241–248, 1982.

[32] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.

[33] J. Sun, G. Peterson, and O. Storaasli. Sparse matrix-vector multiplication design on FPGAs. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '07, pages 349–352, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. R. Thistle and B. J. Smith. A processor architecture for horizon. In *IEEE conference on Supercomputing*, Supercomputing '88, pages 35–41, 1988.

[35] J. Villarreal, A. Park, W. Najjar, and R. J. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *IEEE International Symposium on Field Programable Custom Computing Machines*, FCCM'10, pages 127 –134, 2010.

[36] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[37] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos. FPGA vs. GPU for sparse matrix vector multiply. In *Field-Programmable Technology, 2009. FPT 2009. Int. Conf. on*, pages 255 –262, December 2009.

[38] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based esl synthesis system. *High-Level Synthesis: from Algorithm to Digital Circuit*, 2008.

[39] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays*, FPGA '05, pages 63–74, New York, NY, USA, 2005. ACM.