Proc. Int. Conf. on Compilers Architectures and Synthesis for Embedded Systems (CASES), September 29 - October 4, 2013, Montreal, Canada.

Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads

Robert J. Halstead and Walid Najjar
Dept. of Computer Science
University of California, Riverside
Riverside, California 92507
rhalstea@cs.ucr.edu najjar@cs.ucr.edu

Abstract—**Hardware supported multithreading can mask memory latency by switching the execution to ready threads, which is particularly effective on irregular applications. FPGAs provide an opportunity to have multithreaded data paths customized toeach individual application. In this paper we describe the compiler generation of these hardware structures from a C subset targeting a Convey HC-2ex machine. We describe how this compilation approach differs from other C to HDL compilers. We use the compiler to generate a multithreaded sparse matrix vector multiplication kernel and compare its performance to existing FPGA, and highly optimized software implementations.**

I. Introduction

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. Hardware supported multithreading, as in the MTA architecture [1], can mask memory latency by switching to ready threads and without relying on a cache memory architecture. The rapid context switching between independent threads increases the concurrency which is, by Little's Law, the product of bandwidth and latency.

In FPGA-based code accelerators a frequently executed code segment, 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. The vast majority of applications benefiting from FPGA acceleration have been of regular nature, most of them relying on streaming data. Two common features of such applications are high spatial (streaming data) and temporal (sliding window) locality. Most of the C to HDL tools that have been developed to generate FPGA accelerators, from high-level languages, (such as ImpulseC [2], ROCCC [3], etc.) specifically target such applications.

Irregular applications exhibit very poor locality and hence would not be ideal candidates for FPGA acceleration. However, a multithreaded data path can be implemented on an FPGA with a data path customized to each specific application. Such an execution achieves a high level of efficiency, through the customization of the data path, as well as a high level of parallelism, through the masking of latency. In this paper we describe the implementation of a new tool whose objective is to generate multithreaded data path on FPGAs from C for irregular applications.

The main contributions of this paper are:

- A taxonomy of irregular applications and the constraints they pose on the compiler.
- A compiler implementation generating multithreaded data paths that support irregular applications with dynamic workloads; to our knowledge the first such compiler.
- Demonstration of the compiler using sparse matrix vector multiplication (SpMV).
- An experimental evaluation of the compiler on the Convey HC-2ex.

The rest of this paper is organized as follows: Section II provides background information on multithreaded architectures, irregular applications, and SpMV. Section III shows the related work. Section IV is a discussion on how irregular applications with dynamic workloads can be compiled from C to VHDL. An explanation of how a SpMV kernel is compiled is given in Section V to show the compilers applicability. Section VI describes a series of experiments and results including a comparison to current FPGA, and Software designs. Finally conclusions are given in Section VII.

II. Background

A. Multithreaded Architectures

Graph and sparse linear algebra algorithms are two examples of a growing category of irregular applications that exhibit very poor or no spatial or temporal locality and hence cannot benefit from modern cache architectures. Multithreaded architectures offers an alternative paradigm for improving their performance.

Among the early multithreaded architecture designs is the Horizon architecture [4] which used 256 custom processors connected to global memory. Studies showed an average of 50 to 80 clock cycles for a memory access, but most all of the requests could be fulfilled within 128 cycles. With this knowledge the Horizon's custom processors were built to support 128 concurrent threads, and could context switch in one cycle. Whenever a thread made a memory request the latency could be masked by having the processors handle other ready threads.

In the 1990s the Tera Corporation, later Cray, developed such a design as a commercial machine. The Tera MTA [1], [5], [6] 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 topoly 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 [7] machine, increased the number of supported processing cores to 8192, and the clock frequency to 500 MHz.

B. Sparse Matrix Vector Multiplication

Because of its poor locality, Sparse Matrix Vector multiplication (SpMV) is a memory bound application exhibiting a low Flop/Byte ratio. Its performance depends on architectures continually suppling their cores or accelerators with relevant matrix and vector data. Multiple storage formats were developed to improve cache and memory performance. Some of the more common include: coordinate (COO), compressed sparse row/column (CSR / CSC), and ELLPACK. COO uses three arrays to store only the non-zero elements (NZEs). One array holds the sparse array elements values, another holds their row positions, and the last holds the column positions. CSR and CSC improve the memory utilization by compressing either the row or column array. Using CSR as an example, the column, and value arrays are sorted by row number, and all elements from one row are kept adjacent to each other. The row array is compressed and points only to the start position of each row, which reduces the overall memory utilization. ELLPACK was developed for vector machines. It zero pads rows to improve memory accesses, but it is done at the cost of extra computations and storage. Other formats have been explored and tested. We point the interested reader to [8] for further details.

SpMVs importance has created many customized approaches for software architectures and GPUs. Software optimizations like blocking a matrix at the thread, register and cache levels, or pipelining, or prefetching were explored in [9]. These optimizations were done for five architectures: two AMD, and one Intel out-of-order superscalar architecture, the Sun Niagra2, and the IBM STI Cell machine. The performance of different sparse matrix formats on GPUs is explored in [10]. The impact of memory structures, such as texture memory, played in GPU performance is evaluated. [11] used the ELL format to block row lengths, which allowed tuning of threads for individual GPUs.

III. Related Work

A. Hardware Compilers, and Tools

A number of High Level Synthesis (HLS) tools have been designed aimed at compiling high-level languages (HLLs), such as C/C++, to hardware description languages (HDLs), such as VHDL, Verilog, SystemC or SystemVerilog. Their objective being to bridge the semantic gap between HLLs and HDLs and make the development of FPGA-based accelerators more accessible to traditionally teained application developers.

HLS compilers such as Impulse C [2], ROCCC [14], and AutoPilot (later Vivado) [12], [15]) work with a subset of C to create custom IP accelerators. These tools do not specify a full system, but assist in creating the individual components of a RTL design. Catapult C [13] supports most C++ syntax and constructs (pointers, classes, templates) when generating custom cores. HLS tools do not need to use established languages. Bluespec [16] uses SystemVerilog to describe hardware at a level above typical HDL languages.

ROCCC [3] is a C to VHDL compiler that targets streaming (regular) applications that can analyzed and for which it generate optimized data paths. It attempts to minimize a kernels outgoing memory requests, minimize its area, and maximize its clock frequency. Developers write the kernel in C and the compiler applies optimizations at two levels each with its own intermediate representation (IR); Hi-CIRRF and Lo-CIRRF [17]. Hi-CIRRF identifies FPGA components, which will be needed during execution. It marks and inserts these into the IR as C macros. Its also responsible for duplicating the kernels logic, depending on user specifications, for parallel execution, such as loop unrollng. Lo-CIRRF generates a data flow graph (DFG) as another IR, which becomes the kernels data path. Control systems are built around the DFG to manage execution flow and memory requests.

The CHAT [18], [19] 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 has a deterministic workload. In this work we extend these capabilities to kernels where the workload is non-deterministic.

B. SpMV on FPGA

Current FPGA architectures are ideally suited for applications with streaming data. One approach is to locally store the vector and stream in the sparse matrix data. While modern high-end FPGAs have large on-chip memories in the form of BlockRAM (BRAM) [20] these are not sufficient for storing large data structures as required by HPC applications. The FPGA used in this work, the Xilinx Virtex6 LX760, has about 26 MBits of BRAM on-chip. However, all the BRAM space is not available for the user application, some is used for interfacing to memory and other support functions. As heterogeneous architectures, such as the Convey HC-1/HC-2 [27], become widely available, research is shifting toward end-to-end implementations. In [28] a SpMV personality is develop for a HC-1. In the design all matrix and vector data is stored in global memory which the FPGA accesses through multiple channels. The design caches memory requests locally in case data needs to be reused, but SpMV can be very irregular. The paper reports performance results that are up to 40% of the HC-1's peak. Their design uses 32 individual engines which can produce a peak of 9.6 Gflop/sec.

Substantial work, in implementing sparse matrix operations on FPGAs, has focused on the floating-point Multiply Accumulate (MAc) engine. Since that engine is heavily pipelined, to achieve a high clock rate, it presents a challenge when an unknown number of values must accumulated. Some of the more common designs are outlined here. Adder tree structures [21], [22] with a feedback loops are used to handle multiple row elements in one cycle. Here the number of non-zero elements within the row must be larger than the number of channels into the adder tree. If this is not the case zero padding is typically used to adapt the row size. Other approaches are

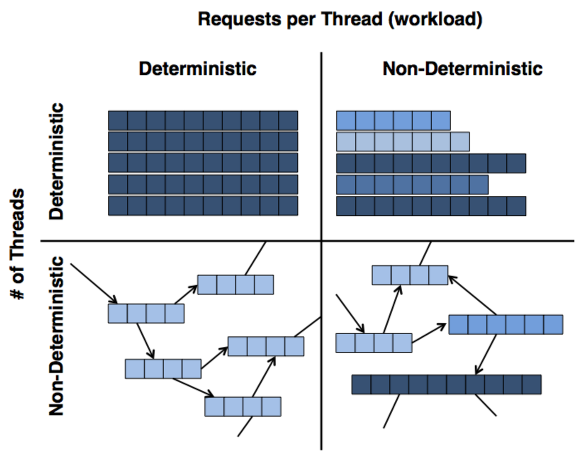

Fig. 1. A taxonomy of irregular applications where the number of threads and the workload size are deterministic or not.

based on statically assigning partial dot-products to multiple processing engines [23], [24]. A control unit is used to manage the communication, and ensure proper execution. Here MAcs may be limited by the number of matrix rows they can manage, typically one or two rows at a time. However, in [25] multiple MAcs are routed onto one FPGA which allows support for more rows in parallel. Research in [26] lead to an accumulation reduction circuit that can handle an arbitrary number of rows. A control unit arbitrates the data flow between the floating-point addition unit and temporary buffers.

IV. Compiler Design

This section describes the CHAT compiler's design. It provides a taxonomy for parallel irregular applications. The compiler's semantics for non-deterministic workloads is discussed, and data path construction for irregular applications with dynamic workloads is explained.

A. A Taxonomy of Irregular Applications

The CHAT compiler builds multithreaded kernels for parallelizable irregular applications. These parallel irregular applications can be further classified by the hardware structures needed for execution. Four classes are proposed as shown in Figure 1. 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 compiled to reduce, or eliminate redundant execution. Applications with a non-deterministic number of threads need FPGA constructs to prevent redundancy. Consider breadth first search. During graph traversal nodes A and B could both point to node C. Without synchronization two threads could be generated to processes node C. Performance would decrease exponentially as each thread then generates more redundant threads.

Applications with a deterministic workload, number of memory requests per thread, can assign threads in round robin fashion to Processing Elements (PEs). Applications with non-deterministic workloads can have variable memory requests. Stalls could occur with round-robin assignment because of

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

unbalanced threads. A Thread Management Unit (TMU), local to the FPGA, can ensure a balanced execution by dynamically assigning threads as PEs become available. This paper explores the compilation of custom TMUs for FPGA kernels.

B. 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 (array) 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 2(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 and 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 2(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 2(b) a threads workload is defined by the outermost for-loop's body.

C. 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

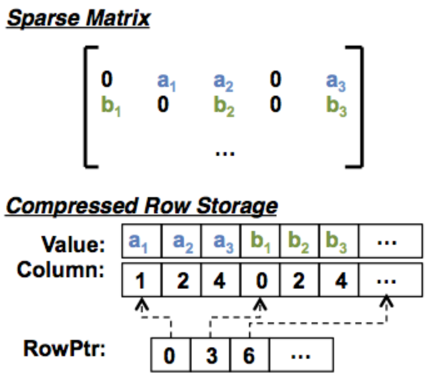

Fig. 3. Compressed Sparse Row (CSR) format for sparse matrix representation with three arrays.

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 available PE. This maintains a balanced execution. A Thread with a disproportionally 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 additionnal threads from being assigned. When all PEs are busy the TMU backloads 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 2(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.

V. Multithreaded SpMV Example

As a proof of concept we use the compiler 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.

```
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. 4. The source code used to generate a custom multithreaded SpMV kernel.

A. Compressed Sparse Row

Multiple matrix formats can be used to store sparse matrices to memory, and each has its tradeoffs. Compressed Sparse Row (CSR) is a very common representation used by both FPGA and software developers. The format stores only relevant matrix information in memory, which limits the unnecessary work done by accelerators and processors. The main draw for CSR is that independent, and hence parallel, threads can be easily identified and generated.

Figure 3 shows a matrix stored in CSR format. All the matrice's non-zero elements are kept in the $Value$ array. Another array, of the same size, is used to store the column positions of the values: the $Column$ array. A one-to-one relationship exists between data points with matching index positions. CSR requires a unique data arrangement for both these arrays. All elements corresponding to the same row must be adjacent to each other, and the elements for row i must come before elements for row $i+1$. It is not required for elements within a row to be sorted by their column values. The third array in CSR, called the $RowPtr$ array, is used to delineate the row elements. Each value in this array represents the starting position of a row. Two adjacent values can be used by a thread to request and processes all elements within a single row of the matrix.

B. SpMV Kernel Code

Sample code for a SpMV kernel is shown in Figure 4. 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 finishes. The designer can unroll the outer for-loop to generate multiple PEs yielding higher parallelism.

C. Convey HC-2ex

Convey Computers [27] 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 coprocessors while the HC-1ex and

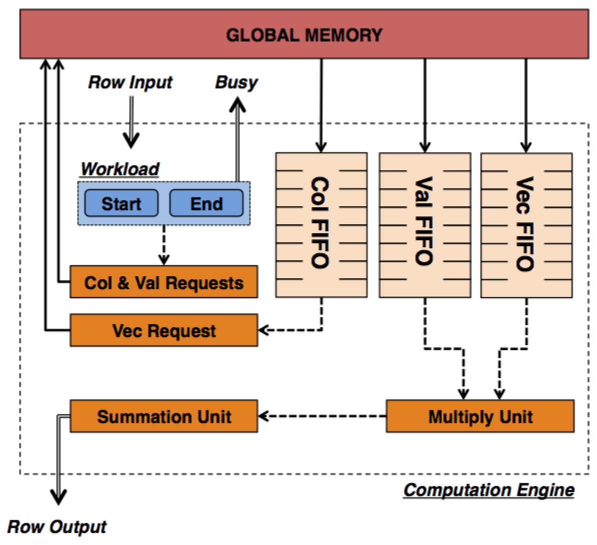

Fig. 5. Each PE is assigned a thread. It requests the necessary data (Column, Vector, and Value) from global memory. Returned data values are pushed through the multiply pipeline, and summation unit.

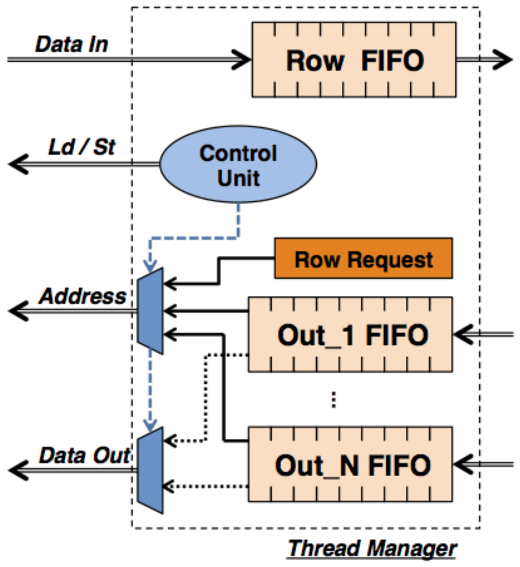

Fig. 6. Thread Management Unit: the output write data and row requests are combined into one memory channel. A control unit handles the conflicts when multiple resources need to use the channel.

HC-2ex machines use four Virtex-6 LX760 FPGAs. The main draw is the shared global memory space [29], [30]. Memory allocated by the host processor can be directly accessed by the FPGA. Memory space is divided between the CPU processors, and the FPGA coprocessor, and performance could depend on where the memory is allocated. With 128 GB per region, its large enough that the CPUs can setup new jobs while the FPGA finishes processing others. A coprocessor has four large FPGAs, which are called Application Engines (AEs). 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 [31]. Thus each AE has 16 memory ports which deliver a peak bandwidth of 19.2 GB/s. The coprocessor's peak bandwidth is about 80 GB/s.

D. 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 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 major components of a PE are shown in Figure 5. 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 yet been returned. Workloads are balanced across PEs because they do not accept new threads until all requests are issued. FIFO buffers within each PE are large enough to support the outstanding memory requests. As memory returns the data for the column array it 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. As reported in Section III-B SpMV reductions circuits have been widely studied. Our compiler uses a circuit similar to the one described in [26]. 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.

E. Thread Management

The Convey HC-2ex has 4 Virtex 6 LX760 FPGAs, called application engines (AEs). Figure 7 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 the designs 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

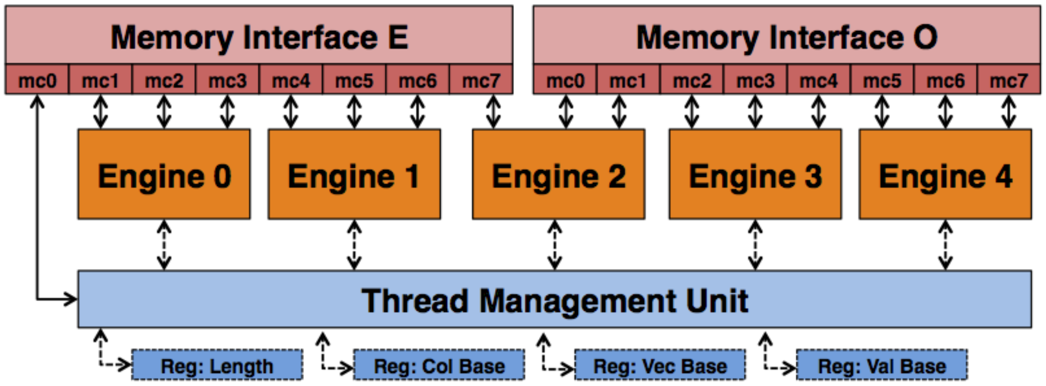

Fig. 7. The MT-FPGA architecture on one AE. Control signals specify the number of jobs (length), and the base addresses of the sparse matrix arrays. All memory channels of the AE are utilized.

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 values from the row pointer array as described in Section V-A. 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 in round robin fashion.

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 6. Read and write request conflicts are resolved by a control unit in favor of the write data, which prevents a deadlock.

F. FPGA Implementation

Integrating a kernel into the HC-2ex requires all memory requests to communicate with Convey's memory interface. Designs are placed and routed varying the number of kernel PEs. Area utilization (including the wrapper) for a single AE is shown in Table I. The design uses only one third of the available slices and BRAMs because it is limited by the memory channels. As discussed in the above each PE requires 3 channels, and the TMU interleaves its requests though the remaining channel.

TABLE I. FPGA UTILIZATION WHEN VARYING THE NUMBER OF PEs.

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%)

VI. Experimental Results

In this section we describe our experimental methodology, and report results for the compiled multithread FPGA kernel (MT-FPGA) on a Convey HC-2ex. Experiments consider scalability, and throughput. MT-FPGA is compared to a cache based FPGA kernel, and optimized multicore CPU implementations.

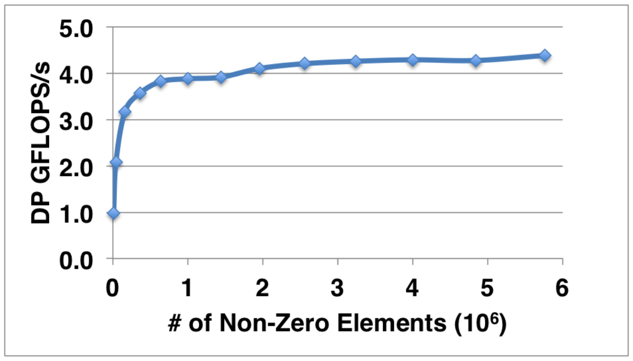

Fig. 8. Sustained Gflop/s on a dense matrix as function of the number of non-zero elements.

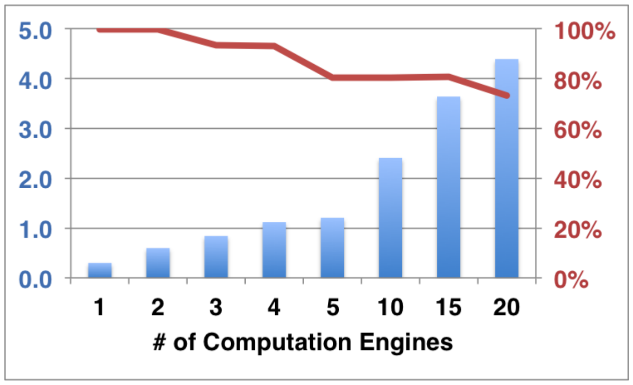

Fig. 9. Sustained Gflop/s (bar) and sustained % of memory throughput (line), on a 2K x 2K dense matrix, as the number of PEs is increased.

A. Experimental Setup

Chosen benchmarks are taken from the University of Florida Sparse Matrix Collection [32]. Table II outlines the relevant information about each matrix: its dimension, the number of non-zero elements, the ratio of non-zero elements to the row size, the sustained performance (DP Gflop/s) and

TABLE II. CHARACTERISTICS OF THE BENCHMARKS MATRICES USED AND THE ACHIEVED PERFORMANCE ON THE MT-FPGA.

Sparse Matrix	Application Domain	Rows	non-zero	nnz/row	DP Gflop/s	% of Peak
Dense	n/a	2,000	4,000,000	2,000	4.34	72%
dw8192	Electromagnetic	8,192	41,746	5.10	1.73	28%
epb1	Thermal	14,734	95,053	6.45	2.50	41%
raefsky1	Fluid Dynamics	3,242	294,276	90.77	3.32	55%
psmigr_2	Economics	3,140	540,022	171.98	2.66	44%
scircuit	Motorola circuit	170,998	958,936	5.61	2.20	36%
torso2	2D model of a torso	115,967	1,033,473	8.91	3.38	56%
mac_econ_fwd500	Macroeconomic Model	206,500	1,273,389	6.17	2.30	38%
cop20k_A	Accelerator cavity design	121,192	1,362,087	11.24	0.60	10%
cant	FEM cantilever	62,451	2,034,917	32.58	3.64	60%
mc2depi	Markov Model	525,825	2,100,225	3.99	2.00	33%
pdb1HYS	1HYS Protein Bank	36,417	2,190,591	60.15	3.43	57%
consph	FEM spheres	83,334	3,046,907	36.56	3.68	61%
nd25k	2D / 3D problem	72,000	14,393,817	199.91	3.27	54%
cage15	Directed Graph	5,154,859	99,199,551	19.24	3.61	60%
Average					2.89	48%

the efficiency (% of Peak) are for the MT-FPGA architecture. Benchmarks are chosen for direct comparisons with the results reported by other approaches [28], [9]. Extra benchmarks are included, nd25k and cage15, to evaluate our approach on very large matrices. Matrices vary in size from 41 thousand to nearly 100 million non-zero elements (NNZ). Irregularity within a matrix varies from a few non-zero elements per row to hundreds of non-zero elements per row. The ”Dense” benchmark is a dense matrix stored in the CSR format. We use it in Section VI-B to eliminate spurious performance behavior during our baseline measurements. By using a dense matrix we remove all irregularity from the benchmark and it is considered a reference point when measuring the sustained throughput.

Performance is reported as Double Precision (DP) GFLOPS/s[1] using all 20 PEs on all four AEs. Each PE is capable of two floating-point operations per cycle, with 20 PEs at 150 MHz the peak rate is 6 Gflop/s. Efficiency is reported as the % of peak performance achieved.

$$(2 * nnz - nrows)/(execution\_time) \tag{1}$$

$$(2 * nnz)/(execution\_time) \tag{2}$$

When computing the FLOPs/s in this paper we use Equation 1. All non-zero elements in the matrix must be multiplied by a vector element; nnz multiplications. Each row must sum all these products;, hence $nnz - nrows$ additions. However, Equation 2 is used by some papers to report throughput. Any results using Equation 2 throughout this paper will be explicitly stated.

B. Sustained Throughput

The MT-FPGA is designed to scale to large problem sizes that require a large number of memory requests, which can be used to mask latency. Early stages of the execution will be dominated by memory requests until the kernel's TMU can buffer extra threads. This startup cost will be aggregated over the execution time, and its effect will be lessened with larger matrices. However, this cost can negatively effect smaller matrices. To determine a ”large enough” matrix we run a series

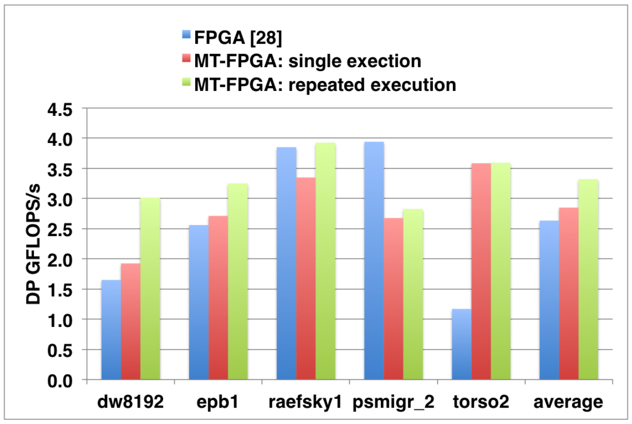

Fig. 10. DP Gflop/s comparison of the cache based design in [28] and the MT-FPGA architecture. Since the benchmarks used in [28] were small, we repeat the execution to achieve close to 1 million non-zero elements.

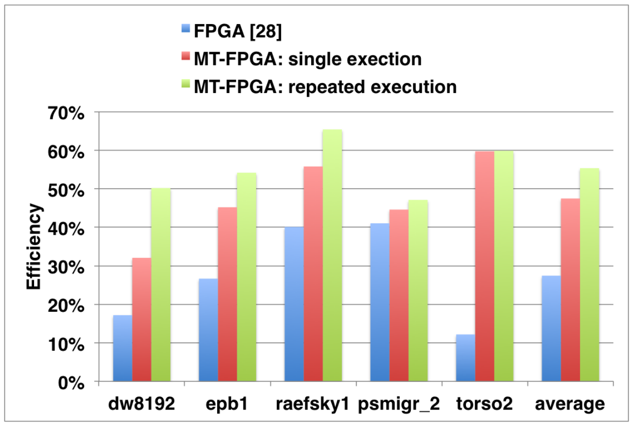

Fig. 11. Efficiency comparison of the cache based design in [28] and MT-FPGA architecture. The design in [28] uses 32 PEs across all FPGAs compared to the 20 for the MT-FPGA.

of test with dense matrices stored in CSR format. By using dense matrices irregularity during execution or in the memory system can be eliminated, which yields a more accurate startup cost estimate. All tests fully utilize the Convey HC-2ex; 20 PEs across all 4 AEs. Matrices vary in size from 10 thousand non-zero elements (NZEs) to 5.76 million NZEs. Results are shown in Figure 8. Sustained throughput quickly approaches 4 GFLOPs/s with 1 million NZEs, and then it slowly rises to

[1]In the rest of the paper all references to floating-point operations are in double precision.

a peak of 4.5 GFLOPs/s. Overall MT-FPGA sustains 75% of its peak throughput. Based on these results 1 million NZEs is the point where matrices are "large enough" to be unaffected by the startup costs.

To study how well MT-FPGA scales and how well the memory system copes with a large number of memory requests, we run a series of test varying the number of PEs. Starting with a single AE the number of engines is increased from one to five, and once full the number of AEs is increased up to four. All tests use a dense 4 million element (2000 x 2000) matrix. The results are shown in Figure 9. The bar chart reports the sustained GFLOPs/s, and the red line reports the percentage of peak throughput sustained. The efficiency with a small number of engines is very close to optimal, which suggests the number of memory requests is not high enough to saturate the memory. However, it slowly drops to 80% as one AE is filled with five PEs. The design scales linearly from one (5 PEs) to three AEs (15 PEs), but drops again slightly with the fourth AE. These results shows that MT-FPGA will scale well at 80% efficiency so long as the memory system can handle the requests.

C. MT-FPGA vs. FPGA caching

The compiled MT-FPGA kernel is compared to the FPGA architecture described in [28]. Both use Convey machines with the same memory system. Peak memory bandwidth (80 GB/s), and clock frequency (150 MHz) are identical. The only difference is with the FPGA used. [28] use a Convey HC-1 (Virtex 5 LX330), and MT-FPGA uses a larger HC-2ex (Virtex 6 LX760). SpMV is a memory bound application, and both FPGAs offer sufficient size for their designs.

MT-FPGA uses memory masking to fully utilize the memory bandwidth. [28] takes a different approach, which relies on caching matrix and vector data locally to the FPGA. Caches are structured, with efficient line sizes, for an equal number of reads to each of the memory's SGDIMMs. Vector caches are maintained for each processing element (PE), and a matrix cache is shared across all PEs within an AE.

Throughput comparisons are done with the matrices chosen in [28]. However, these matrices were typically small. Only 2 were larger than 500 thousand NZEs, and only one of those was above 1 million NZEs. Because of this two sets of tests were run for MT-FPGA. The first test measured throughput on a single execution of the matrix. The second test did repeated executions to compensate for the smaller sizes. The number of runs were repeated until the number of NZEs was above 1 million elements. For example the $dw8192$ matrix has 41,746 NZEs. It was run 24 times resulting in 1,001,904 NZEs. Figure 10 reports the throughput comparison results. Results are copied directly in [28] which used Equation 2. For equal comparison MT-FPGA results use the same equation and are slightly higher than those reported in Table II.

When executing a matrix one time MT-FPGA averages a higher throughput across all benchmarks, and with multiple runs MT-FPGA delivers a higher throughput for all but one benchmark. The exception is $psmigr\_2$, which is more densely packed than the other matrices and benefits more from caching. Because our approach targets large matrices we consider $torso2$ case (the only one above 1 million elements) most telling. Here performance drops dramatically for the caching architecture while MT-FPGA sustains 3.4 Gflop/s of 56.7% of peak.

MT-FPGA achieves better performance with fewer PEs (20 compared to 32 in [28]), which results in much higher efficiency. Measurements are shown in Figure 11. Peak throughput for MT-FPGA is 6 GFLOPs/s while the caching approach peaks at 9.6 GFLOPS/s. MT-FPGA's results for the single execution case average 47%, and its results for repeated execution average 55%. The best performing benchmark for [28] achieves 41% efficiency, and the average for all benchmarks is 27%.

D. MT-FPGA vs. Multicore

In this section we compare the performance of MT-FPGA with the multicore architectures reported in [9]. Table III reports important architecture information about the multicore machines. Before running the experiments MT-FPGA was not expected to outperform the multicore architectures, but results are included for comparative purposes. The goal for this paper is compiling custom multithreaded FPGA kernels, and the focus was not strictly on performance. However, the multicore machines targeted performance, and were customized for each architecture. They run at much higher clock frequencies (GHz compared to a FPGA's MHz), and have more dedicated hardware. However, MT-FPGA did fair well with the multicore architectures. It beat a few CPUs in raw throughput, and offered comparable throughput to most of the others.

This was achieved in spite of the CPUs being highly optimized. Software optimizations are done at three levels. The first set of optimizations targets kernel parallelization. Execution threads are blocked to ensure an evenly distributed workload. In the NUMA and Cell architectures threads are statically assigned to minimize communication. The second set of optimizations target the kernel's data structures. The sparse matrix vectors, and the result vector are blocked according to the cache line. To reduce memory requirements register blocking is used where adjacent non-zero elements are joined together into one set of coordinates. Heuristics are used to improve processing time. Matrices with less than 64K columns use 16 bit indexes instead of 32, or 64 bits. Finally low-level kernel optimizations are considered. A custom kernel is used for each architecture. The Cell processor's kernel uses software pipelining to mask the instruction latency, and it also uses branchless execution. Explicit software prefetching is used in the out-of-order processors. Other common loop optimizations are also performed.

Sustained throughput is shown in Figure 12. MT-FPGA achieves an average throughput higher than the AMD Santa Rosa and Intel Clovertown CPUs. Its throughput is comparable to the AMD Barcelona and the Sun Niagara 2, but could only sustain about half the throughput of the STI Cell machine. However, with the high clock frequency and dedicated hardware those machines sustain a small fraction of their overall throughput. Peak throughput for the multicore machines reneges from 17.6 to 74.7 GFLOPs/s while the FPGA peaks at only 6 GFLOPs/s. Figure 13 shows the percentage of peak performance each machine achieves. MT-FPGA's efficiency is over 30% in seven of the eight benchmarks, over 50% in

TABLE III. ARCHITECTURE SPECIFICATIONS FOR THE CPUS AND GPU COMPARED TO IN THIS PAPER.

Architecture	Cores (Sockets)	Clock (GHz)	Peak (DP GFLOPS/s)
AMD - Opteron2214 (Santa Rosa)	4 (2)	2.2	17.6
AMD - Opteron 2356 (Barcelona)	8 (2)	2.3	73.6
Intel - Xeon E5345 (Clovertown)	8 (2)	2.3	74.7
Sun - Niagara 2	16 (2)	1.16	18.7
STI - Cell QS20	16 (2)	3.2	29

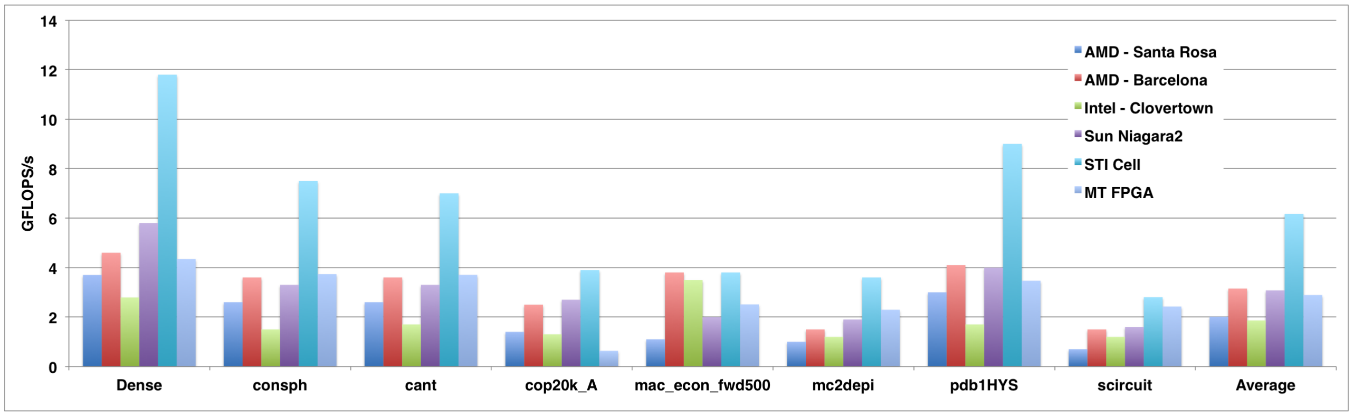

Fig. 12. Sustained throughput (DP Gflop/s) for multicores CPUs, and MT-FPGA.

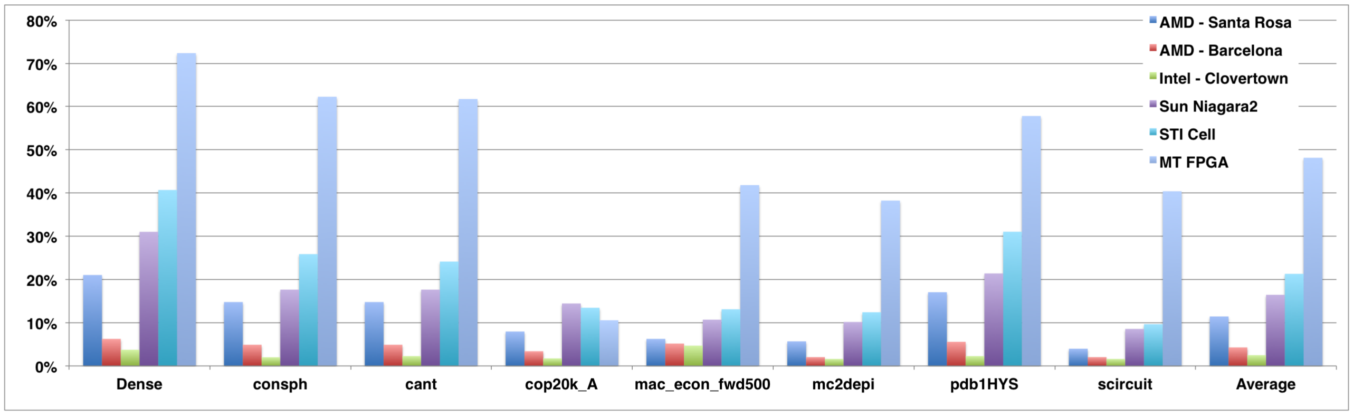

Fig. 13. Efficiency (reported as % of peak) for multicores CPUs, and MT-FPGA.

four and averages 48%. The highest efficiencies are achieved by the STI Cell and the Sun Niagara with averages between 15% and 25% which are less than half that of MT-FPGA. As heterogeneous FPGA architectures, which are in their infancy, improve the efficiency may or may not change, but the throughput potential likely will.

VII. CONCLUSION

The multithreaded execution model, by rapidly switching to ready threads, allows the masking of long memory latencies. It can achieve a high degree of concurrency, equal to the product of the memory latency and the number of outstanding memory requests. For irregular applications that suffer from very poor spatial and temporal locality, this model can alleviate the impact of memory latency while increasing parallelism.

FPGAs provide an opportunity to customize a data path to each individual computation hence achieving a higher efficiency, albeit at a lower clock speed. Traditionally, FPGA-based accelerators have been used for highly regular streaming applications.

In this paper we presented a compiler that generates customized multithreaded data paths on FPGAs for irregular applications with dynamic workloads. We outline how a thread management unit is created, and how it dynamically balances threads across multiple processing elements. Results show that multithreading as a FPGA paradigm is valid for irregular applications. Using a Sparse Matrix Vector multiplication as a proof of concept, we show the MT-FPGA kernel outperformed a cache based FPGA kernel in both overall throughput and efficiency. Compared to highly optimized multicore architectures the MT-FPGA kernel has higher or comparable throughput to four tested architectures, and about half the throughput of the STI Cell architecture. However, efficiency is more than double that of the closest multicore architecture. The multithreaded design scales to very large matrices. On matrices over 3 million non-zero elements it achieved 50% efficiency and over 3 GFLOPs/s throughput for each. However, performance is limited by startup costs on matrices smaller than one million elements.

Acknowledgments

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

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," in *Proceedings of the 4th International Conference on Supercomputing*, ser. ICS '90. New York, NY, USA: ACM, 1990, pp. 1–6.

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

[3] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Field-Prog. Custom Comp. Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, may 2010, pp. 127 –134.

[4] J. T. Kuehnand and B. J. Smith, "The horizon supercomputing system: architecture and software," in *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 28–34.

[5] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in *Proceedings of the 6th International Confonference on Supercomputing*, ser. ICS '92. New York, NY, USA: ACM, 1992, pp. 188–197.

[6] 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*, ser. Supercomputing '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–8.

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

[8] Y. Saad, *SPARSKIT: A basic tool kit for sparse matrix computation*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[9] 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*, vol. 35, no. 3, pp. 178 – 194, 2009.

[10] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA, Tech. Rep., 2008.

[11] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *Proceedings of the 5th Int. Conf. on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Springer-Verlag, 2010, pp. 111–125.

[12] 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.

[13] "Catapult C," *http://www.mentor.com/*.

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

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

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

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

[18] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," in *Proceedings of the first workshop on Irregular applications: architectures and algorithm*, ser. IAAA '11. New York, NY, USA: ACM, 2011, pp. 31–34.

[19] R. J. Halstead and W. Najjar, "Compiling irregular applciations for reconfigurable systems," in *International Jounal on High Performance Computing and Networking*, to appear.

[20] "Virtex-6 family overview," *http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf*, 2012.

[21] 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*, ser. FCCM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 349–352.

[22] 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*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 63–74.

[23] 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*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 75–85.

[24] 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*, June 2010, pp. 64 –70.

[25] 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*, December 2009, pp. 255 –262.

[26] 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*, August 2009, pp. 287 –292.

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

[28] 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*, may 2011, pp. 1 –8.

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

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

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

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