Compiling irregular applications for reconfigurable systems

Robert J. Halstead*

Department of Computer Science and Engineering,
University of California,
Riverside, CA 92507, USA
E-mail: rhalstea@cs.ucr.edu
*Corresponding author

Jason Villarreal

Jacquard Computing,
Riverside, CA 92507, USA
E-mail: jason@jacquardcomputing.com

Walid A. Najjar

Department of Computer Science and Engineering,
University of California,
Riverside, CA 92507, USA
E-mail: najjar@cs.ucr.edu

Abstract: Algorithms that exhibit irregular memory access patterns are known to show poor performance on multiprocessor architectures, particularly when memory access latency is variable. Many common data structures, including graphs, trees, and linked-lists, exhibit these irregular memory access patterns. While FPGA-based code accelerators have been successful on applications with regular memory access patterns, they have not been further explored for irregular memory access patterns. Multithreading has been shown to be an effective technique in masking long latencies. We describe the compiler generation of concurrent hardware threads for FPGAs with the objective of masking the memory latency caused by irregular memory access patterns. The CHAT compiler extends the ROCCC toolset to generate customised state information for each dynamically generated thread. Initial results show a speed-up of 50x.

Keywords: irregular memory; custom hardware accelerated threads; CHAT; compilers; field programmable gate array; FPGA; C to VHDL; irregular applications; reconfigurable systems.

Reference to this paper should be made as follows: Halstead, R.J., Villarreal, J. and Najjar, W.A. (2014) 'Compiling irregular applications for reconfigurable systems', *Int. J. High Performance Computing and Networking*, Vol. 7, No. 4, pp.258–268.

Biographical notes: Robert J. Halstead is currently a PhD student at the University of California, Riverside. He is pursuing research in the fields of reconfigurable computing and HPC. In particular his focus is on compiler technologies to ease development of FPGA-based designs. He also focuses on heterogeneous FPGA systems.

Jason Villarreal is a Senior Engineer at Jacquard Computing, Inc. He received his PhD in Computer Science from the University of California, Riverside in 2008. His research interests include reconfigurable computing, compiler optimisations for hardware constructs, and hardware acceleration of high performance computing applications.

Walid A. Najjar is a Professor in the Department of Computer Science and Engineering at the University of California, Riverside. He received his PhD in Computer Engineering from the University of Southern California in 1988. His research interests are in the fields of computer architecture and compiler optimisations, embedded systems and sensor networks. Lately, he has been very active in the area of compilation for FPGA-based code acceleration and reconfigurable computing. NSF, DARPA and various industry sponsors have supported his research.

1 Introduction

Algorithms exhibiting irregular memory access behaviour have been notoriously difficult to speed-up because of their poor data locality. It is not possible to co-locate data and computation on the same node and data caches cannot be efficiently exploited. Multithreaded architectures, such as the Tera MTA (Alverson et al., 1990, 1992; Snavely et al., 1998), are designed to mask memory long latency by rapidly switching between concurrent threads; hence, they are ideally suited for irregular applications.

Field programmable gate arrays (FPGAs) are well known for their speed and efficiency on regular algorithms that operate on massive datasets. In an FPGA-based hardware acceleration the most frequently executed computation(s) is synthesised as a customised data path through which the large amount of data is streamed. Applications that have been demonstrated to benefit from FPGA acceleration include signal and image processing, computer vision, data mining, bioinformatics, financial analysis, etc. All of these applications exhibit regular memory accesses.

In this paper we introduce and describe *custom hardware accelerated threads* (CHAT) a novel approach to multithreading on FPGAs. CHAT compiles the code to a custom circuit on the FPGA, however, when a thread performs a long latency memory operation its state is saved to a queue of waiting threads and the execution switches to another ready thread. Upon the return of the data from memory, the relevant thread resumes its execution. By overlapping execution with memory access CHAT masks the memory latency thereby achieving a high throughput and speedup. The CHAT implementation builds upon the ROCCC compiler toolset (Villarreal et al., 2010; ROCCC, http://roccc.cs.ucr.edu/). The compiler generates the necessary hardware structures to synchronise the results with the related threads. Initial experimental evaluation on the convey computers HC-1 (Brewer, 2010) shows a speed-up higher than 50x over software using a single Xilinx Virtex 5 LX330 FPGA.

The paper is organised as follows: Section 2 summarises the related work. A classification of irregular applications is given in Section 3. Section 4 describes the CHAT compiler framework, and toolset. Sample applications are presented and explained in Section 5. The experimental results, run on the convey HC-1 machine, are reported and discussed in Section 5.2.

2 Related work

2.1 Multithreaded architectures

In the late 1980s research into multi-processor systems with large shared memory was being conducted. The Horizon architecture (Kuehnand and Smith, 1988; Thistle and Smith, 1988) was built with 256 custom processors. Research showed an average of 50–80 clock cycles per memory accesses, and most all memory request were completed within 128 cycles. The processor in the Horizon architecture was thus built to manage the state information for 128 concurrent threads. Hence it can support up to 128 outstanding memory requests masking the memory latency caused by having 256 processors sharing a common memory.

In the early 90s the Tera Corporation, starting from the experience acquired with the Horizon machine, built the Tera MTA (Alverson et al., 1990, 1992). The MTA design consisted of 256 processors sharing 64 GB of memory organised as a distributed NUMA architecture. Its interconnection network allowed better scaling to a larger number of processors. It also forced instruction requests through a shared cache lowering the network traffic. Custom processors supported the issuing of one memory request per thread per cycle. The maximum memory latency from any processor to any memory module was 128 cycles. Each processor could support up to 128 active threads. The MTA design (Snavely et al., 1998) was later evolved into the Cray XMT (Feo et al., 2005). While the MTA had only 256 processors the XMT machine could support up to 8,192 processors, but the largest ones built had 512 processors. The shared memory was also increased from 1 TB to 128 TBs for the MTA, and the clock speed was improved from 220 MHz to 500 MHz.

2.2 Heterogeneous platforms

The 1980s also saw reconfigurable fabrics being integrated into large supercomputers. These include a large number of cutting edge CPUs coupled with a number of FPGA devices with full or partial sharing of memory. Notable among these is the Cray XD1. The Cray XD1 evaluated by the Naval Research Laboratory (Osburn et al., 2006) consisted of 432 dual-core processors with 144 Virtex-II FPGAs and six Virtex-4 FPGAs. The machine consisted of 150 nodes each with one FPGA, two processor cores and 8GBs of shared memory. Of these, 144 had one Virtex-II, and another six had one Virtex-4 FPGA.

The convey computers HC-1 (Brewer, 2010) is the first heterogeneous machine to support cache coherent shared virtual memory accesses from both the software (CPU execution) and the hardware (FPGA execution). This virtual memory allows an application to switch execution between software and hardware. Without the need to offload data this switch can be made with little overhead. The HC-1 has four Virtex-5 LX330 FPGAs, further allowing multiple sections of an application to be written to a FPGA without need of reconfiguration at runtime. In the HC-1ex the Virtex 6 LX760 is used instead of the Virtex 5.

2.3 Parallel irregular applications

The CHAOS (Das et al., 1994) runtime system is a set of libraries developed for parallel irregular applications in the mid 90s. It analyses the indexes into array access to break loops into smaller sections which can be optimised individually. It then generates an inspector to manage memory, and communication with other processors. Ideally

the inspector and portioning is done once so execution time is amortised over the application's lifetime.

The LocalWrite (Han and Tseng, 2000) approach developed for shared memory processors architectures works to identify mutually exclusive datasets in an application. This helps minimise the replicated buffers, and eliminate any synchronisation when writing back. However it may require re-computation of edge data elements shared between datasets.

For applications with more complex irregular access, patterns libraries like KeLP (Fink et al., 1998) can help designers manage them. After segmenting the application into blocks the designer can manipulate how the blocks interact, and control their communication schedules. KeLP then generates the low-level data structures.

2.4 High level synthesis tools

Many commercial and open source high-level synthesis tools, like CatapultC (http://www.mentor.com/), ImpulseC (http://www.impulseaccelerated.com/), ROCCC (http://roccc.cs.ucr.edu/) and Zhang et al. (2008), have appeared since the early 2000s. They can improve an application's performance by identifying parallelism in existing code, or they may require programmers to identify and rewrite the parallel regions. Because the programming methodologies for software and hardware are so different these tools may limit a high-level language's functionality when specifying hardware regions. This can be done by accepting a subset of the language's constructs, or by restricting how certain constructs can be used. For example, the tool may accept pointers, but not allow dynamic memory allocation, or dynamic indexing.

These language limitations are acceptable in these tools because of the target applications they were built for. Most of these HLS tools are optimised for streaming applications which usually exhibit regular memory accesses. But, CHAT's goal is to extend these HLS tools to irregular applications. When analysing the DFG the CHAT compiler identifies and classifies dynamic memory accesses where typical tools will give an error.

3 Irregular applications

Irregular applications exhibit unstructured patterns in the access of data in memory: consecutive memory reads have no or very little correlation to previous reads. The poor temporal and spacial locality cause a large number of cache misses hindering the applications performance. In this section we attempt to classify irregular accesses patterns into categories based on discernible knowledge of the number of threads and the number of memory accesses per thread at compile time. Based on this knowledge the compiler can then generate a custom FPGA kernel for the given application. The application category helps determine the type of components needed for the kernel.

3.1 Determinable threads and determinable memory requests

In the first category of irregular applications the compiler can determine the number of threads, and the number of memory requests per thread. These values are set at the kernel's initialisation. Kernels of this form are typically the simplest irregular applications. An example of this type of application is given in Algorithm 1. The compiler can statically infer that *size_i* elements will be written to the *result* array, and thus *size_i* threads will be needed at runtime. All threads will perform the same operation, and always need *size_j* elements from array *B*. In this way the number of threads, and memory request are known once the kernel is initialised with *size_i* and *size_j*.

Algorithm 1 The compiler can determine both the number of threads needed and the number of memory accesses per thread

```
int *A, *B, *result;
for (int i = 0; i < size i; ++i)
    for (int j = 0; j < size j, ++j)
        result [i] = op (A, B [j], i);
```

Algorithm 2 The compiler can determine the number of threads needed, but not the number of memory accesses per thread

```
int *A, *B, *C, *result;
for (int i = 0; i < size i; ++i)
    for (int j = 0; j < C [i]; ++j)
        result [i] = op (A, B [j], i);
```

Algorithm 3 The compiler can determine the number of memory accesses per thread, but not the number of threads

```
while (! terminate) {
    for (int i = 0; i < size i; ++i)
        process (i);
}
```

Algorithm 4 Simplified BFS algorithm

```
Queue Q;
while (! Q. empty ( )) {
    process (Q. to p ( ));
    Q. pop ( );
}
```

Note: The compiler cannot determine either the number of threads needed, or the number of memory accesses per thread.

3.2 Determinable threads and undeterminable memory requests

In the second category of irregular applications the compiler can determine the number of threads, but the number of memory requests per thread cannot be determined until runtime. Again the number of threads is set at the kernel's initialisation, but each thread's memory accesses are not specified by an initialisation variable. An example of this type of application is shown in Algorithm 2. The application will require *size_i* threads, but each thread requests *C*[*i*] elements from array *B*. Because each thread has a unique number of memory requests the kernel must manage a unique state for each thread.

3.3 Undeterminable threads and determinable memory requests

In the third category of irregular applications the compiler can determine the number of memory accesses a thread will make, but cannot determine the number of threads that will be needed. Having a static memory access pattern for a thread can allow the compiler to optimise the outgoing requests. Possible optimisations could be sending multiple requests per cycle, or buffering data to reduce the number of accesses. Algorithm 3 shows how applications in this category could be written. The application runs until its termination condition is met, and each iteration will accesses the data with a known pattern.

3.4 Undeterminable threads and undeterminable memory requests

In the fourth category of irregular applications the number of threads, and the number of memory requests is undeterminable at compile time. This is the most general category of irregular applications. They typically execute over dynamic data structures like trees or graphs using pointers to determine the next node. Consider the breath first search (BFS) algorithm on a graph, as shown in Algorithm 4. Each node in the graph would be a thread of execution, and the nodes adjacency list is the number of memory requests needed for the thread. Graph data structures do not always provide a size, and BFS may not require searching the entire graph. Because of this the kernel will not know how many threads are needed, or how many memory requests each thread will require. As the kernel processes one thread it will be dynamically creating new ones.

4 The CHAT compiler

The *CHAT* compiler is designed to help developers better implement irregular applications on FPGAs using a multithreaded execution model. At a high level CHAT is a C to VHDL compiler. Applications are specified in C, and the compiler generates a custom kernel that manages all the necessary memory request, and data-path components. CHAT's goal is to help developers by abstracting away low level implementation details such as the synchronisation within a thread, context switching, and low-level HDL design. Allowing the developer to focus on the application's functionality. Thus, it helps improve productivity by reducing the development time. Another goal is easy portability between architectures. To do this CHAT provides a simple, but general interface to the generated VHDL for requesting memory.

4.1 Execution model

The fundamental execution model for CHAT it to use concurrent threads in hardware. The execution of a thread is suspended following its memory accesses, and resumed when the accessed data is available. When a thread is suspended the execution switches to the next available thread from a queue of ready threads. Thread states are kept locally on the FPGA allowing this context switch to be done in one cycle. To cope with memory access latency CHAT generated kernels provide support for *multiple outstanding memory requests*. In this model it is assumed that the memory system can support multiple outstanding memory requests and that they are returned in the *requested order*. In some instances, the memory system supports *multiple concurrent memory channels*, these can be either physical or virtual.

CHAT analyses the C application to identify the memory access patterns, and determine the kernel's main functionality. Based on this functionality a pipelined data path is generated. Memory accesses are classified as regular or irregular based on static analysis of their indexes. Custom components generated for regular accesses can request memory with no data from a thread's state. Irregular access components generate data-paths that read thread state data to request memory.

4.2 Thread model

In CHAT each output value is assigned to its own hardware thread. All memory requests needed to generate the output are part of this thread. These memory requests are not always unique to a thread. They can be shared to reduce the kernel's overall memory accesses. As an example, memory requests to *B* in Algorithm 1 are shareable between threads. The management of all threads is done by the custom architecture generated by CHAT.

CHAT's custom architectures are meant to be portable across many platforms. Including platforms with long memory latencies which are not common for most FPGA applications. To cope with a long overhead CHAT kernels will processes multiple threads in parallel, and issue multiple memory requests from them to mask latency. CHAT merges the memory accesses it can at compile time, but this does not prevent independent threads from requesting identical memory locations. It is the developers responsibility to account for this when designing their application. However, a cache could be placed outside the CHAT kernel to mitigate these types of applications, but the

designer must add the cache by hand. A single thread may not require enough memory accesses the fully mask the latency. For this reason CHAT kernels support multiple thread execution by the custom components. This is done by saving thread states in the components.

By statically analysing the C code CHAT determines the number of memory channels and the dependencies between them. With this a data flow for the threads can be created. Threads request data from the controllers, and to insure full utilisation of the memory channels threads are queued in FIFOs and wait to be processed.

If needed, the data from the requests will be saved in the FIFOs as part of the thread's state. Data can be saved until the thread's computation data-path, or in the case of irregular memory requests until a memory controller needs it. We can use FIFOs to hold thread states because all threads will follow the same data flow. If a memory channel is ever shared between multiple threads then the request is stalled until all threads are ready to read it.

As an example, after compiling Algorithm 1 CHAT would generate a kernel similar to Figure 1. We can assume the operation is compiled into a datapath, and we can also assume the operation requires memory locations $A[B[j]]$. Note two address generation units (AGUs) will be created. One for A which is irregular and one for B that is a regular request unit. A given thread will need $size_j$ requests from both B and A. The thread can begin requesting data from B immediately, and the returned data will be queued in a FIFO. As data from B becomes ready A reads it and begin issuing its own memory requests. Returned data for A is queued in another FIFO for the datapath to read. Because memory requests are independent they can be issued in parallel, and the thread's data is kept between two FIFOs. The threads current state can be determined by the state of FIFOs A, and B.

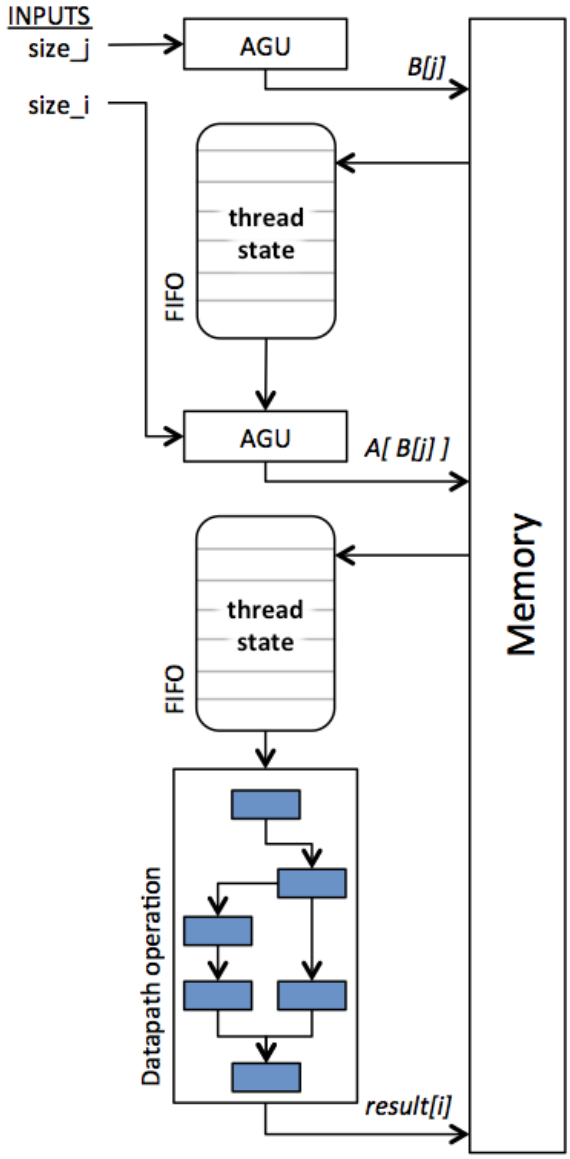
4.3 Performance optimisations

The overall performance of a kernel is greatly affected by the efficiency of the data-path, and the amount of concurrent execution it can achieve. So, fully utilising the FPGA is a priority. However, performance is also dependent upon the specific FPGA being used. With the wide array of FPGA options available to developers it is impractical to have the compiler optimise for any FPGA. Thus, CHAT allows the developer to customise the generated kernel to their specific FPGA.

User specifiable parameters to the CHAT allow a developer to unroll a kernel, better utilising the area of the FPGA. Unrolling also increases the number of concurrent thread data-paths a kernel can execute. It would also increase the number of memory channels needed for a design. However, analysis of unrolled designs can identify identical memory channels and combine them into one shared memory channel. CHAT manages all synchronisation between the threads, and memory channels.

For portability and performance developers can design their own FIFO implementations. This is done because most FPGAs offer custom on-chip BRAMs which may be preferred for FIFOs over LUT-based implementations. However, the FIFOs must adhere to a specific interface, and functionality.

Figure 1 CHAT kernel example showing regular and irregular memory request component (see online version for colours)

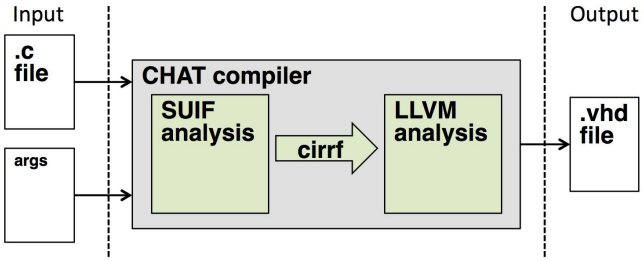


Notes: An AGU is created for both memory channels. B is a regular request from 0 to size_j, and A is an irregular request using data requested by B.

4.4 Implementation of CHAT

CHAT is implemented using the ROCCC compiler's (Villarreal et al., 2010; ROCCC, http://roccc.cs.ucr.edu/) infrastructure. CHAT does analysis at two levels as shown in Figure 2. High level analysis is done using the SUIF 2.0 toolset (Wilson et al., 1994) which builds a data flow graph (DFG) and generates an intermediate representation (IR), this IR is then further optimised with low level analysis using the LLVM compiler (Lattner and Adve, 2004).

Figure 2 CHATs design flow (see online version for colours)



ROCCC is a C to VHDL compiler toolset designed specifically for the generation of FPGA-based code accelerators. Its distinguishing features are its extensive set of compiler transformations and optimisations. ROCCC was not designed to create hardware for entire applications, but instead focuses on the critical regions of large software systems. The critical regions typically consist of a loop nest performing extensive computation on large amounts of data. Hence, the ROCCC code takes advantage of the extensive amount of parallelism available on FPGAs and the ability to implement large computational pipelines on streams of data while attempting to minimise off-chip memory fetches and control flow, which are better handled on microprocessors. Among the ROCCC design goals: maximise throughput, minimise memory accesses, minimise the size of the generated circuit, support code reuse through the import of modules in C, VHDL or as IP cores, generate platform independent code and support fast design space exploration. In its current design ROCCC supports codes that have memory accesses whose order is compile-time determinable. These can be in one, two or N dimensional arrays. CHAT extends the ROCCC compiler to support irregular memory accesses. This extension is currently designed for the convey HC-1 but could be extended to other platforms that can support multiple outstanding memory request and where masking memory latency can be beneficial.

The high-level analysis of an application is implemented using the SUIF toolset which generate the application's DFG. The DFG's components consist of the memory channels (marked as regular or irregular), and the main computation data-path for the application. Regular memory components in the DFG store the start, end, and stride elements for memory requests. Irregular memory components store what thread state data should be used, and how it should be processed. If the design is unrolled all components are first duplicated in the DFG. Then any components that are duplicates will be merged into one component in the graph. Finally, the DFG is encoded in a cirrf and passed to the next stage of the CHAT compiler; low-level analysis and implementation.

Low level implementations are done with custom passes in the LLVM compiler. Each component from the DFG is assigned to a basic block, and subsequent passes evaluate it at this level. The thread state FIFOs are added to the DFG to create a control flow graph (CFG). CHAT does not specify how the FIFOs are built. It assumes dual clocked FIFOs with no fall-though, and a specific port list, but the implementation is left to the designer. FIFOs can thus be implemented fully in logic, or with board specific constructs. With all FIFOs inserted in the CFG the kernel components are generated in VHDL. Synchronisation logic is added to the kernel ensuring proper execution. This step ensures a thread state is not read if all components needing it are not ready. It also ensures a component stops requesting if the FIFO it must write to is full.

The implementation of CHAT differers from ROCCC in a few major ways. First, ROCCC assumes that all memory accesses follow a regular pattern. We extend the framework to detect and support irregular memory access in CHAT. Focusing on irregular applications expands the set of applications compilable onto reconfigurable fabrics. However, compiling these applications requires the removal of certain optimisations that were possible when assuming only regular accesses. An example of this would be smart buffering of memory requests. The final kernel format for CHAT also diverged from the standard ROCCC format. ROCCC kernels have three main components; an input controller, an output controller, and the data-path. All incoming memory channels are managed by the input controller, and likewise for the output controller. Always having three components allows a simple implementation for the global system synchronisation that ROCCC uses. ROCCC manages a global state machine to ensure proper kernel execution. ROCCC only moves the thread state forward once all necessary data elements are ready. This model would hinder performance in irregular kernels because the kernel uses more data-paths than just the main computation one. Each irregular memory controller has its own data-path. Generating a global state manager for an arbitrary number of data-paths quickly becomes a difficult task. For this reason CHAT removes the global memory controllers used in ROCCC in favour of many smaller decoupled memory controllers for each channel. Allowing each component to manage its own state, and send its output data to the next component in the data flow when it is done.

5 Experimental evaluation

In this section we explore the performance potential gained with memory masking on the FPGA vs. software. CHAT generates the irregular memory hardware which is attached to the optimised data paths generated by ROCCC. This allows for kernels that can operate on a subset of data from a larger dataset like processing SQL queries, or sparse algebra operations. Most kernels already supported by ROCCC are also supported by CHAT with a few exceptions, namely systolic array generation. However, when measuring performance we want to minimise any improvements from data path optimisation. We do this by studying kernels with memory access bottlenecks. Two kernels are shown in Algorithms 5 and 6: both of them sum values over arbitrary columns of a matrix. They differ by which columns are selected per row. Algorithm 5 will always choose the same columns while Algorithm 6 can

have unique columns per row. Both kernels are executed in hardware and software on the convey computers HC-1.

Algorithm 5 Summation with 1-dimensional index stream

```
void summation (int **A, int *B, int *C, int m, int p) {
    int i, j;
    for (j = 0; j < m; ++j)
        for (i = 0; i < p; ++i)
            C [j] += A [j] [B[i]];
}
```

Algorithm 6 Summation with 2-dimensional index stream

```
void summation (int **A, int **B, int *C, int m, int p) {
    int i, j;
    for (j = 0; j < m; ++j)
        for (i = 0; i < p; ++i)
            C [j] += A [j] [B[j][i]];
}
```

While kernels that fit into the known threads and known memory access category are typically easy to implement by hand they are a necessary first step into the compilation of more complex kernels. This paper outlines the management of threads where everything is known at compile time. Future work will consider multithreading on kernels where this is not the case. Cases, like SpMV or graph traversal, where parts of the kernel are unknown until runtime.

The convey uses Intel Xeon CPUs for software processing. These are not the fastest processors available, but are still orders of magnitude faster than the FPGAs. The CPU clock frequency is 2.13 GHz, while the FPGA clock is limited to 150 MHz. The CPU processors are acceptable for our purpose because the application's bottleneck will be memory access time. The global memory of the HC-1 is shared between both hardware and software making it ideal to test our memory masking approach.

5.1 Evaluation kernels

5.1.1 One dimensional indexing

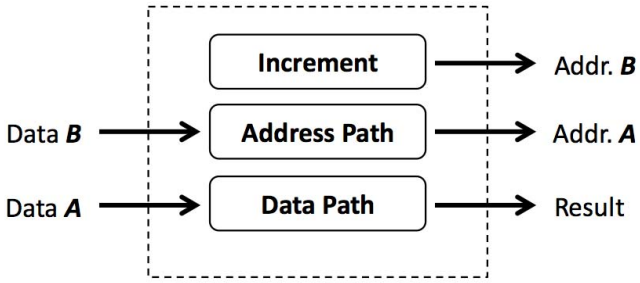
The kernel shown in Algorithm 5, and expressed by Equation 1 will sum specific columns of a given row in a matrix. Through the kernel's execution each row will sum the same columns, and the selected columns are specified by array B which is a one dimensional array. Each thread in the kernel will be the summation for a single row. When unrolling the design to process multiple rows concurrently the data read from memory channel B will be shared between all executing threads. Thus, CHAT will generate a kernel that uses only one memory channel to access B, and synchronising it among all thread data-paths.

$$C[m] = \sum_{i=1}^{p} A[m, B[i]] \tag{1}$$

The kernel as a whole takes a two dimensional array $A[m, n]$ of values, and a list of indexes $B[p]$ as input, where all values in B are less than n. The application runs each row, m, of A through its data path summing the values in the columns specified in B and storing each result into the corresponding element of C.

For Algorithm 5 each thread data-path created has three major components shown in Figure 3. The increment component is a regular memory generator. At runtime the number of columns per row to sum is specified by p, and the incriminator requests the index into B for each of these columns. The index into A is dependent upon the value returned from B making it an irregular memory access. The address path component tracks the current row, m, and uses the index provided by B to generate a request into A. The data-path is a simple summation that takes the values returned to A and produces the result which then gets written back to memory.

Figure 3 Summation of a single row



Notes: Values are first fetched from B. Processing these values determines a new location in A. The value stored at this location is accumulated into the final result. This is repeated for each row in A.

5.1.2 Two dimensional indexing

The kernel presented in this section is similar to the one dimensional kernel presented above, but instead B is a two dimensional array. This allows the summation of independent columns on a per row basis. With each row summing different columns unrolled designs can no longer share a memory channel. The design, as written, requires each row to still select the same number of columns. This limitation in the kernel places it in the first category of irregular applications where the total number of threads, and memory access are known for each data-path.

5.2 Experimental results

In this section we describe in more detail the convey computers HC-1 system where we performed our tests. We provide a comparison between our kernels and their software implementations. We use one of the CPUs, and compared it to only one of the FPGAs on the HC-1. Finally we discuss the affects shared streams have on performance of the kernels.

5.2.1 Convey computers HC-1 platform

The convey computers HC-1 system (Brewer, 2010) is the first heterogeneous machine to support cache coherent shared virtual memory accesses from both the software (CPU execution) and the hardware (FPGA execution). This virtual memory allows an application to easily switch its execution between software and hardware. The HC-1 has eight 2.13 GHz Xeon CPU cores and a hybrid co-processor with four application engines (AEs) consisting of four Xilinx Virtex 5LX330 FPGAs1. An outline of the HC-1's FPGA and memory interface layout is shown in Figure 6.

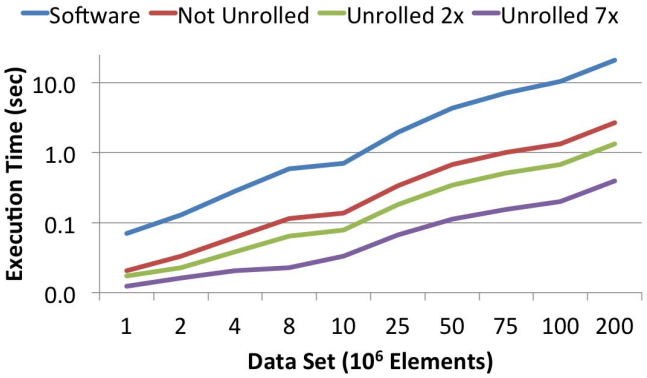
The whole system consists of two 1U chassis, the host motherboard and the co-processor. The four AEs interface to memory via eight full duplex memory controllers, each implemented on a Virtex 5LX150, through a full crossbar itself implemented in FPGAs as well. The cumulative peak bandwidth to memory is 80 GB/s when the data is uniformly distributed across all memory modules (20 GB/s per AE). The interface to memory can be run at 300 MHz or 150 MHz. The latter allows for 16 memory channels per AE which increases parallelism but with a fixed bandwidth. Each channel is 64-bits. The memory interface, implemented with the Intel Front Side Bus (FSB) protocol, is fully cache coherent (snoopy protocol) with the host memory allowing for one global shared address space. Unlike PCI-based FPGA co-processors, the memory coherence on the HC-1 is fully transparent and does not require explicit data movements to maintain coherence with the host data. The data returned from memory, within memory channels, can be returned in order. This accomplished by reorder buffers in the memory controllers.

An application engine hub (AEH) consists of two FPGAs. One is the co-processor interface to the host FSB, it implements the snoopy coherence protocol and maintains the page tables for the co-processor. The other FPGA implements a soft processor that is the effective host of the co-processor. It is connected to each AE, it loads the programming file for AE and every call to an accelerator code on an AE is issued by the soft processor. The accelerator code that runs on the AEs is developed with the convey personality development kit (PDK) consisting of a set of Verilog programmes and makefiles. The PDK defines all the non-programmable components of the accelerators, such as the crossbar and the memory controllers. It supports hardware/software co-simulation (with Mentor Graphics ModelSim) of the accelerator code. In addition to the accelerator code, a hardware wrapper is instantiated by the PDK on each AE that implements the memory interfaces. It occupies about 20% of the BRAMs and 10% of the slices on each FPGAS of the HC-1 (much smaller percentage on the HC-1ex). The accelerator code clock frequency is fixed at 150 MHz. Only one of the four AEHs is used for results in this paper, and is compared to one CPU processor.

5.2.2 Runtime, speedup and utilisation

The execution time for the 1-dimensional kernel is shown in Figure 4 with a logarithmic scale. Note that The FPGA is reported for three distinct cases; no unrolling, 2x unrolling and 7x unrolling. The execution for all three cases is approximately constant, at about 10 msec, up to a dataset size of ten million elements. This is where the startup and initialisation costs of the HC-1 accelerators dominate the total execution time.

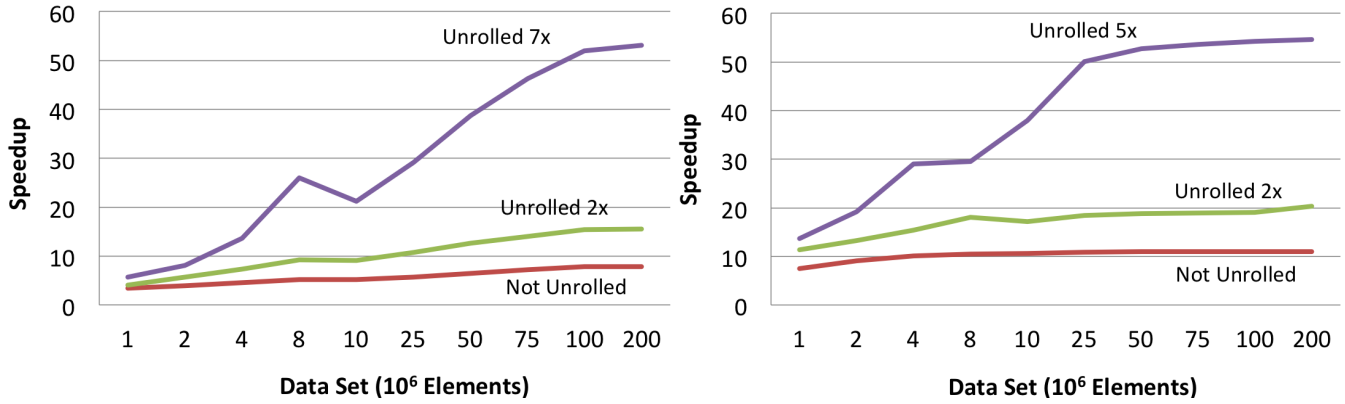
Figure 4 Execution time (sec), on a logarithmic scale, of 1-dimensional hardware kernels and their software equivalent (see online version for colours)



Each accelerator FPGA on the HC-1 has 16 memory channels. In the 1-dimensional kernel the *B* occupies a channel and the *A* and *C* arrays are unrolled up to seven times. In the 2-dimensional kernel each instance of *A*, *B* and *C* requires a separate channel so the maximum unrolling is 5x.

Figure 5(a) shows the speedup achieved by the first kernel over software. Note that the speedup with *no unrolling* is 8x over software, due to the ability of the multithreaded model to mask memory latency. Because the index stream *B* is shared each initial response to *B* produces a memory request from *A* equivalent to the unroll factor. When fully unrolled (at 7x) a single response to *B* will result in seven requests for *A*. This is the reason behind the 50x speedup over software when fully unrolled, and 15x speedup when unrolled by a factor of 2x. Execution time is lowered from 21 seconds in software to 1.3 seconds when unrolled by 2x, and under 0.5 seconds when fully unrolled on the largest dataset.

Figure 5 Speedup over software achieved, varying the dataset size, (a) 1-dimensional index stream B (b) 2-dimensional index stream B (see online version for colours)



(a) (b)

Figure 6 Convey HC-1 architecture (see online version for colours)

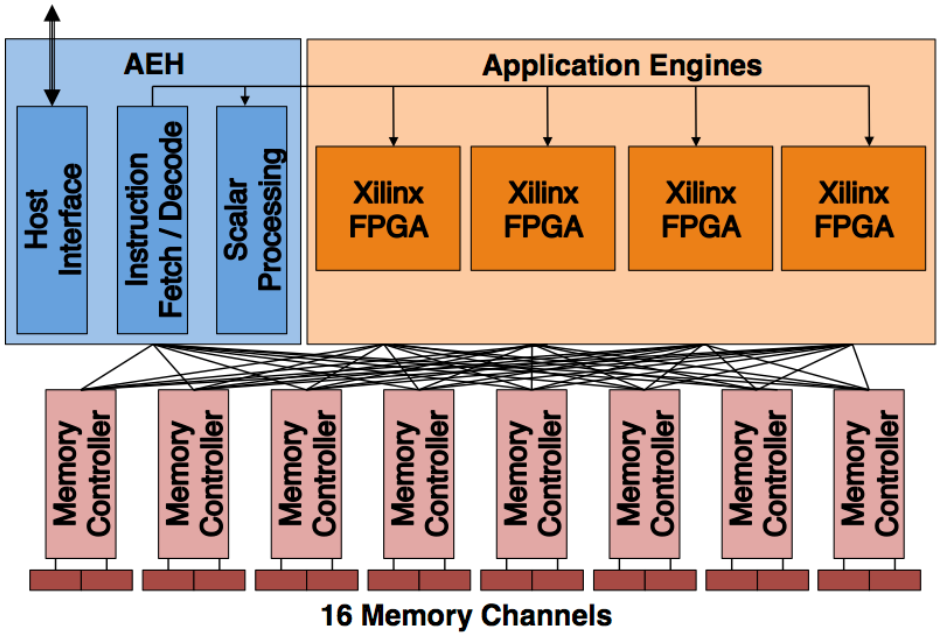
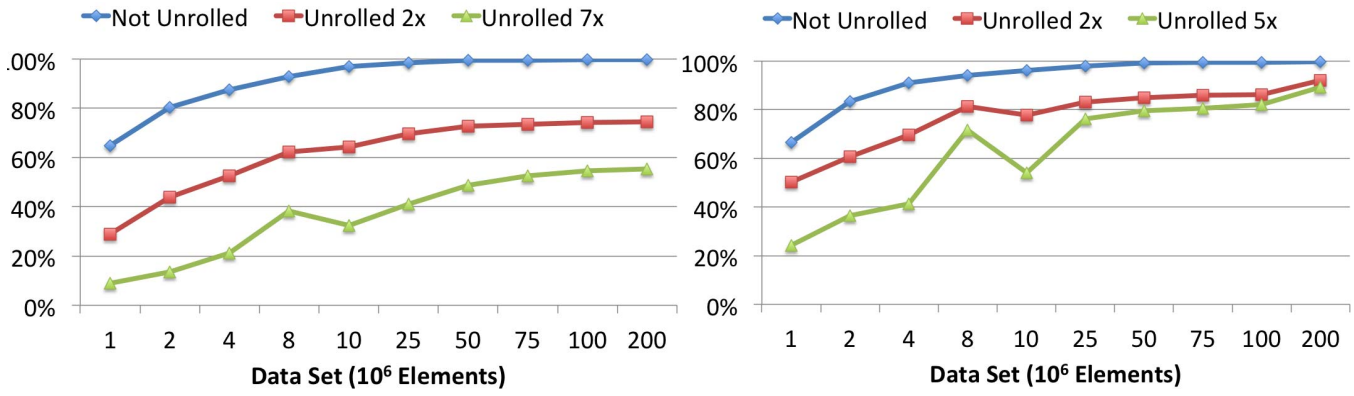


Figure 7 Utilisation of memory channels (%) versus the dataset size, (a) utilisation in 1-dimensional index stream kernel (b) utilisation in 2-dimensional index stream kernel (see online version for colours)



(a) (b)

The speedup results for the 2-dimensional kernel are shown in Figure 5(b). Note that this kernel does not benefit from the sharing of the index stream *B*. This results in poorer caching performance in the software execution and a higher speedup for the FPGA execution with no unrolling (10x). Other observations are that the dip is at four million elements and is less pronounced than the first kernel. The maximum speedup is achieved earlier than the previous kernel (at 50 million elements versus 100 million).

In Figures 5(a) and 5(b) we can observe a discontinuity in the speed-up curve around dataset sizes with eight million elements. This is probably due to the distribution of the data across the memory modules that can result in collisions when the pressure on the memory system is high as it is with 7x and 5x unrolling respectively.

The *utilisation* is another performance measure we have evaluated. In this context we define utilisation as the percentage of cycles a memory channel is busy reading memory, aggregated over all the memory channels used. It is shown in Figures 7(a) and 7(b). The following observations can be made: the not unrolled code quickly saturates the few memory channels allocated to it; the fully unrolled 1-dimensional kernel does not saturate the memory system achieving just about 50% utilisation; the 2-dimensional kernel rapidly approaches 80% utilisation of the available memory bandwidth.

5.2.3 Area

We report the kernel's area results with the convey interface wrapper attached on a Virtex 5 LX330 FPGA. To guarantee in-order memory accesses, the wrapper has an optional crossbar that was added. Because of this it occupies a large portion of the designs slices. The CHAT framework is BRAM intensive because it uses FIFOs to store all thread states, and data for its designs. This can be seen in Table 1 where the CHAT kernels greatly increase BRAM utilisation, and only slightly increase slice utilisation. The frameworks for 1D and 2D kernels are very similar. The main difference is in the kernel logic, and because of this we see a small increase in slice while BRAM usage is almost identical.

Table 1 Area requirement for both 1, and 2 dimensional kernels on a Virtex 5 LX330 FPGA

Design	*Slices*	*BRAMS*
Convey wrapper	14,620 (28%)	34 (12%)
1D not unrolled	16,651 (32%)	55 (19%)
1D 2 unrolled	17,083 (33%)	59 (20%)
1D 7 unrolled	19,004 (37%)	79 (27%)
2D not unrolled	16,977 (33%)	55 (19%)
2D 2 unrolled	17,235 (33%)	60 (21%)
2D 5 unrolled	19,598 (38%)	75 (26%)

Note: All results are reported with the convey wrapper attached.

6 Conclusions

Because of their lack of data locality, irregular applications have been notoriously hard to parallelise. Multithreaded execution with support for multiple concurrent threads fixed in the hardware at design time have been shown to be a viable approach to the masking of long memory latencies and hence are a good option for irregular applications. In this paper we have introduced *CHAT*, a compilation tool that generates customised hardware support for multithreaded execution on FPGAs. We describe its execution model and implementation and report on the initial stages of its performance evaluation as implemented on the convey computers HC-1. Using just one accelerator FPGA we show a speed-up of up to 50x over a single Intel Xeon on simple irregular kernels.

Acknowledgements

This work has been supported in part by NSF Awards 0905509 and 0811416, by Jacquard Computing Inc. and by the Air Force Research Lab.

References

Alverson, G., Alverson, R., Callahan, D., Koblenz, B., Porterfield, A. and Smith, B. (1992) 'Exploiting heterogeneous parallelism on a multithreaded multiprocessor', in *Proc. of the 6th Int. Conf. on Supercomputing, ICS '92*, pp.188–197, ACM, New York, NY, USA.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B. (1990) 'The tera computer system', in *Proc. of the 4th Int. Conf. on Supercomputing, ICS '90*, pp.1–6, ACM, New York, NY, USA.

Brewer, T.M. (2010) 'Instruction set innovations for the convey HC-1 computer', *IEEE Micro*, March, Vol. 30, No. 2, pp.70–79.

CatapultC [online] http://www.mentor.com/.

Das, R., Uysal, M., Saltz, J. and Hwang, Y.s. (1994) 'Communication optimizations for irregular scientific computations on distributed memory architectures', *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, pp.462–478.

Feo, J., Harper, D., Kahan, S. and Konecny, P. (2005) 'Eldorado', in *Proceedings of the 2nd Conference on Computing Frontiers, CF '05*, pp.28–34, ACM, New York, NY, USA.

Fink, S.J., Baden, S.B. and Kohn, S.R. (1998) 'Efficient run-time support for irregular block-structured applications', *Journal of Parallel and Distributed Computing*, Vol. 50, Nos. 1–2, pp.61–82.

Han, H. and Tseng, C-W. (2000) 'Efficient compiler and run-time support for parallel irregular reductions', *Parallel Computing*, Vol. 26, Nos. 13–14 pp.1861–1887.

ImpulseC [online] http://www.impulseaccelerated.com/.

Kuehnand, J. and Smith, B. (1988) 'The Horizon supercomputing system: architecture and software', in *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing, Supercomputing '88*, pp.28–34, IEEE Computer Society Press, Los Alamitos, CA, USA.

Lattner, C. and Adve, V. (2004) 'LLVM: a compilation framework for lifelong program analysis & transformation', in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, California, March.

Osburn, J., Anderson, W., Rosenberg, R. and Lanzagorta, M. (2006) 'Early experiences on the NRL Cray XD1', in *Proc. of the HPCMP Users Group Conference*, pp.347–353, IEEE Computer Society, Washington, DC, USA.

ROCCC [online] http://roccc.cs.ucr.edu/.

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

Thistle, M.R. and Smith, B.J. (1988) 'A processor architecture for Horizon', in *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing, Supercomputing '88*, pp.35–41, IEEE Computer Society Press, Los Alamitos, CA, USA.

Villarreal, J., Park, A., Najjar, W. and Halstead, R. (2010) '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, pp.127–134.

Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S-W., Tseng, C-W., Hall, M.W., Lam, M.S. and Hennessy, J.L. (1994) 'SUIF: an infrastructure for research on parallelizing and optimizing compilers', *SIGPLAN Not.*, December, Vol. 29, No. 12, pp.31–37.

Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C. and Cong, J. (2008) 'Autopilot: A platform-based ESL synthesis system', *High-Level Synthesis: from Algorithm to Digital Circuit*, Chapter 6, pp.99–112, Springer, Netherlands.