

Towards Compilation of Streaming Programs into FPGA Hardware

Franjo Plavec

Zvonko Vranesic

Stephen Brown

University of Toronto

Department of Electrical and Computer Engineering

10 King's College Road, Toronto, Ontario, Canada

{plavec, zvonko, brown} @eecg.toronto.edu

Abstract

There is an increasing need for automated conversion of high-level design descriptions into hardware. We present a flow that converts a software application written in the Brook streaming language into a hardware description targeting FPGAs. We use a combination of our source-to-source compiler and a commercial C2H behavioral synthesis compiler. Our approach results in a significant throughput increase compared to software and ordinary C2H results (up to 8.9X and 4.3X, respectively). The throughput can be further increased by using more hardware resources to exploit data parallelism available in streaming applications.

1. Introduction

A complete system on a programmable chip (SOPC) can often fit into an FPGA device. Such a system usually contains one or more processing nodes, either soft- or hard-core, and a number of peripherals. As the complexity of SOPCs grows, there is a need for tools that allow users to design their systems at a high level. Major FPGA vendors already provide such tools for building systems based on soft processors, which help software developers who wish to target FPGAs. However, such systems do not exploit the full potential of FPGAs, because they fail to generate circuits that fully exploit the parallel nature of an application. To take advantage of parallelism available in FPGAs has required mastering one of the Hardware Design Languages (HDLs). Recently, there has been a push towards supporting automatic compilation of software programs into hardware.

There are three basic approaches to automatic compilation of software into hardware. Behavioral synthesis compilers [4] analyze programs written in a high-level sequential language, such as C, and attempt to extract instruction-level parallelism by analyzing dependencies among instructions, and mapping independent instructions to parallel hardware units. Several such compilers have been released, including C2H [1] from Altera, which is fully in-

tegrated into their SOPC design flow. The main problem with the behavioral synthesis approach is that the amount of instruction-level parallelism in a typical software program is limited. In the case of C2H, programmers often have to restructure their code and explicitly manage hardware resources, such as mapping of data to memory modules.

Another approach is to take an existing parallel programming model and map a program written in it onto hardware [14]. This approach allows programmers to express parallelism, but they have to deal with issues such as synchronization, deadlocks and starvation.

The third approach is to use a language that allows the programmers to express parallelism without having to worry about synchronization and related issues. One class of languages that is attracting a lot of attention lately is based on the streaming paradigm. In streaming, data is organized into streams, which are collections of data, similar to arrays, but with the elements that are guaranteed to be mutually independent [2]. Computation on the streams is performed by kernels, which are functions that implicitly operate on all elements of their input streams. Since the stream elements are independent, a kernel can operate on individual stream elements in parallel, thus exploiting data-level parallelism. If there are multiple kernels in a program, they can operate in parallel in a pipelined fashion, thus exploiting task-level parallelism. Figure 1 depicts an application consisting of 5 kernels (depicted as circles) and memory buffers that pass stream data between the kernels. Streaming programming languages do not specify the nature of the memory buffers. We use FIFO buffers because of their small size, which allows us to implement them in an on-chip memory in the FPGA.

In this paper we show that the streaming paradigm is suitable for implementation in FPGAs. Our compiler converts kernels into hardware blocks that operate on incoming streams of data. We chose Brook streaming language [2] as our source language because it is based on the C programming language, so it is more likely to be accepted by the programmer community. Brook language has been used

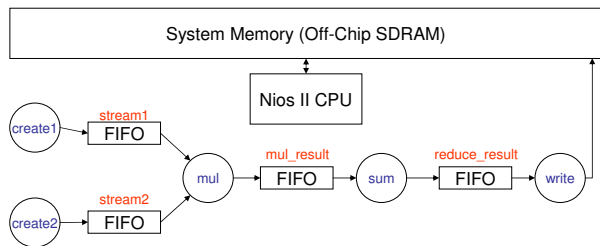


Figure 1. Sample streaming application.

in a number of research projects [5, 11, 13]. We show that a program expressed in Brook can be automatically converted into parallel hardware, which executes significantly faster than software. Our methodology uses our modified version of the Brook compiler, and also leverages the C2H commercial tool.

The rest of the paper is organized as follows. In section 2 we describe related work in the field of stream computing. Our design flow and tools are described in section 3. Section 4 presents experimental results. We make some concluding remarks in section 5.

2. Stream Computing

The term stream computing (a.k.a. stream processing) has been used to describe a variety of systems. Examples of stream computing include dataflow systems, reactive systems, signal processing and other systems [12]. We focus on streaming applications defined through a set of kernels, which define the computation, and a set of data streams, which define communication. This organization allows the compiler to easily analyze communication patterns in the application, so that parallelism can be exploited. When a programmer specifies that certain data belongs to a stream, this provides a guarantee that the elements of the stream are independent from one another. The computation specified by the kernel can then be applied to stream elements in any order. In fact, all computation can be performed in parallel, limited only by the available computing resources.

Stream processing is based on the Single Instruction Multiple Data (SIMD) paradigm, and is similar to vector processing. The major difference between stream and vector processing is in computation granularity. While vector processing involves only simple operations on (typically) two operands, a kernel may take an arbitrary number of input, and produce one or more output streams. In addition, the kernel computation can be arbitrarily complex, and the execution time may vary significantly from one element to the next. Finally, elements of a stream can be complex (e.g. custom data structures), compared to vector processing, which can only operate on primitive data types.

The recent interest in stream processing has been driven by two trends: the emergence of multi-core architectures and general-purpose computation on graphics processing units. Multi-core architectures have sparked interest in

novel programming paradigms that allow easy programming of such systems. Stream processing is a paradigm that is suitable for both building and programming these systems. For example, the Merrimac supercomputer [13] consists of many stream processors, each containing a number of floating-point units and a hierarchy of register files. A program expressed in a stream programming language is mapped to this architecture in a way that captures data locality through the register file hierarchy. Programs for Merrimac are written in the Brook streaming language [2]. Another similar project is the RAW microprocessor and the accompanying StreamIt streaming language [8].

The processing power of graphics processing units (GPUs) has led to their use for general-purpose computing [3]. GPUs are suitable for stream processing because they typically have a large number (16 or more) of 4-way vector execution units as well as a relatively large memory. Streaming languages can be used to program these systems, because kernel computation can be easily mapped to the processing units in a GPU. Various compilers that convert code written in a streaming language into a code suitable for execution on a GPU have been proposed [3, 6]. GPU Brook [6] is a variant of the Brook streaming language specifically targeting GPUs. The GPU Brook compiler hides many details of the underlying GPU hardware from the programmer, thus making programming easier.

2.1. Streaming on FPGAs

Several research projects have investigated stream processing on FPGAs. Howes et al. [7] compared performance of GPUs, PlayStation 2 (PS2) vector units, and FPGAs used as coprocessors. The applications were written in ASC (A Stream Compiler) for FPGAs, which has been extended to target GPUs and PS2. ASC is a C++ library that can be used to target hardware. The authors found that for most applications GPUs outperformed both FPGAs and PS2 vector units. However, they used the FPGA as a coprocessor card attached to a PC, which results in a large communication overhead between the host processor and the FPGA. They showed that removing this overhead improves performance of the FPGA significantly. Their approach does not clearly define kernels and streams of data, so the burden is on the programmer to explicitly define which parts of the application will be implemented in hardware.

Bellas et al. [10] developed a system based on "streaming accelerators". The computation in an application is expressed as a streaming data flow graph (sDFG) and data streams are specified through stream descriptors. sDFG and the stream descriptors are then mapped to customizable units performing computation and communication. The disadvantage of this approach is that the application has to be described in a somewhat obscure format (sDFG).

Our approach to streaming on FPGAs is closest to the

work described in [9], which also converts computation into hardware IP blocks interconnected in a pipelined fashion. Their approach targets ordinary C programs augmented with directives that allow the programmer to specify how an application maps to hardware. Our approach is based on a streaming language, in which parallelism can be expressed without any knowledge of the target hardware.

3. Compiling Brook to Hardware

We propose using a streaming language as a natural choice for software programmers wishing to target their applications to FPGAs. We believe that the stream processing paradigm is suitable for implementation in FPGAs, because programmable logic blocks in FPGAs are suitable for implementation of parallel computation. Also, FPGAs are easily reprogrammable, so the generated hardware can be tailored to the needs of a specific application.

The design space for FPGA implementation of streaming applications is large. For instance, a kernel could be implemented as custom hardware, a soft-core processor, or a streaming processor. In either case, several parallel instances of hardware implementing the kernel may be necessary to meet the throughput requirement. The choice of types and numbers of hardware units will affect the topology of the interconnection network. Finally, stream elements can be communicated through on-chip or off-chip memories, organized as regular memories or FIFO buffers.

We generate custom hardware for each kernel in the application. An ordinary soft processor would be a poor choice for implementing kernels, because it can only receive and send data through its data bus, which may quickly become a bottleneck. Custom hardware units can have as many I/O ports as needed by an application and are likely to provide the best performance. However, if a kernel is complex, the amount of circuitry needed for its implementation as custom hardware may be excessive, in which case a streaming processor may be a better choice. In this paper we focus on implementing kernels as hardware units.

We base our work on GPU Brook [6] because it is open-source, used in many projects and supported through a community forum. To implement a program written in GPU Brook in an FPGA, the kernel code should be converted into an HDL. Instead of performing this conversion directly, we generate C code for each kernel and then use the C2H behavioral compiler [1] to convert the C code into hardware.

All of the C code is generated automatically by our compiler, so the programmer only has to write the Brook source code and pass it through our flow. The first part of the flow is a source-to-source compiler. We reused the original GPU Brook parser and wrote a code generator that emits C code for each kernel. C2H allows functions in the C code to be implemented as hardware blocks. Altera documentation refers to the generated hardware block as a "hardware ac-

celerator" [1]. Hardware accelerators act as coprocessors to the main soft processor (Nios II), which controls the accelerators and executes the code that was not selected for acceleration. Current version of C2H does not support floating-point data type and operations.

Depending on the desired functionality, an accelerator can have one or more ports for accessing other (memory) modules in the system; for each pointer dereference in the original C code, a new port is created. Special pragma statements can be used to define which memories in the system a port connects to. We use this functionality to define how streams are passed between kernels through FIFOs. FIFOs are small so they can be placed on-chip, and they fit naturally into the streaming paradigm, because they act as registers in the pipeline. FIFOs are used instead of simple registers because they provide buffering for cases when execution time of a kernel varies between the elements. For example, consider the system in Figure 1. If the kernel *mul* takes a long time to process one element, the next kernel downstream (*sum*) could become idle if there was just one register between them. Using FIFOs, the *sum* kernel can process data from the FIFO. As long as the *mul* kernel delivers the next stream element before the FIFO buffer becomes empty, the *sum* kernel will not have to stall.

3.1. Example Brook Program

In Brook, streams are declared similarly to arrays, except that characters "<" and ">" are used instead of square brackets. Kernels are denoted using the *kernel* keyword. We illustrate the work done by our compiler using the following Brook code:

```
kernel void mul (int a<>, int b<>, out int c<>) {
    c = a*b; }
reduce void sum (int a<>, reduce int r<>) {
    r = r+a; }
```

The code is incomplete and contains only two kernels: *mul* and *sum*. Kernel code refers to individual streams, not stream elements. This prevents programmers from introducing data dependencies between stream elements. It is assumed that the operation is to be performed over all stream elements. A special kind of kernel, so called reduction kernel, uses several elements of the input stream to produce one element of the output stream. These kernels are used to perform reduction operations, and are denoted by the *reduce* keyword.

To convert Brook code into C, our compiler generates an explicit *for* loop around the statements inside the kernel function to specify that the kernel operation should be performed over all stream elements. For the Brook code above, our compiler produces the code similar to this:

```
void mul () {
    volatile int *a, *b, *c; int _iter;
    for (_iter=0; _iter<IN_LENGTH; _iter++) {
        *c = (*a) * (*b);
```

```

} }
void sum() {
volatile int *a, *r;
int _temp_r, _iter, _mod_iter=0;
for (_iter=0; _iter<IN_LENGTH; _iter++) {
if ((_mod_iter == 0) && (_iter != 0))
*r = _temp_r;
if (_mod_iter == 0)
_temp_r = *a;
else
_temp_r = _temp_r + (*a);
if (_mod_iter == (IN_LENGTH/REDUCE_LENGTH-1))
_mod_iter = 0;
else
_mod_iter = _mod_iter+1;
}
*r = _temp_r; }

```

In this code, all compiler-generated variable names start with the “_” character. For the *mul* kernel, the code is a straightforward *for* loop that reads elements from input streams (FIFOs) *a* and *b*, multiplies them and writes the result to the output stream (FIFO) *c*. Pointers *a*, *b* and *c* are connected to FIFOs, which is specified by C2H pragma statements not shown in the above code. The pragmas are also automatically generated by our compiler. FIFOs are implemented in hardware, so kernel code does not have to manage FIFO read and write pointers. The *IN_LENGTH* limit for the *for* loop was automatically inserted by the compiler, based on the sizes of the streams passed to the *mul* kernel at the place in the code where the kernel was called.

The code generated for the *sum* kernel is more complicated because *sum* is a reduction kernel. The reduction operation can result in more than one element in the output stream. In our example, the input stream with *IN_LENGTH* elements is reduced to a stream of *REDUCE_LENGTH* elements. This means that *IN_LENGTH/REDUCE_LENGTH* elements of the input are summed to produce one element of the output.

Once the C code is generated, it is passed through Altera’s C2H compiler, which generates a Verilog description of hardware accelerators for the kernels. The Verilog code is then passed through Quartus II flow to generate a programming file for the target FPGA.

To evaluate the system’s performance, we generate the streams using simple loops inside the *create1* and *create2* kernels. This approach, as compared to reading the input data from memory, ensures that the runtime of our benchmarks is not dominated by communication with the shared off-chip memory, which would be the case if three different kernels were using the same memory. The results are written to the main memory using a specialized kernel *write*, so that they can be checked for correctness. The system also includes a Nios II processor, which verifies the correctness of the results, and measures execution time. For some applications, such a processor may not be necessary, because the input data may be coming from the outside and the results

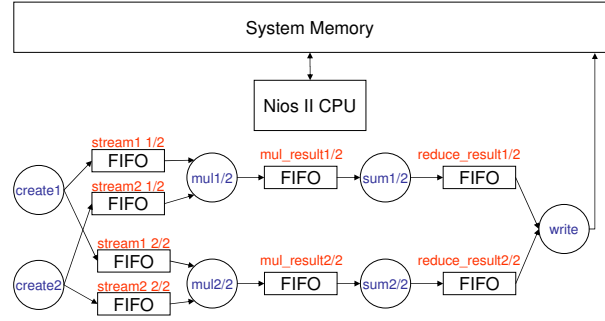


Figure 2. Replication example.

may be passed back to the outside world.

3.2. Exploiting Data Parallelism

The system in Figure 1 contains hardware accelerators, which can operate in parallel, thus exploiting task-level parallelism. However, each accelerator processes stream elements one at a time, meaning that data parallelism is not exploited. One way to exploit data parallelism is to replicate the functionality of each kernel. This is possible because stream elements are independent, so they can be processed in parallel. In theory we could have as many kernel replicas as the number of elements in the streams being processed. In practice, the kernel that is a bottleneck for the application will be replicated as many times as necessary to achieve required throughput. Replication will usually be limited by the available hardware resources.

Figure 2 shows the application from Figure 1 with kernels *mul* and *sum* replicated. In this example, *sum* and *mul* have comparable throughput so both of these kernels have to be replicated to increase the application throughput. Kernels *create* and *write* were not replicated because it is assumed that they already operate at a maximum throughput, limited by either the application or the I/O communication interface. If that is not the case, replicating one of these kernels may be beneficial. In this situation, *create* kernels send elements to *mul1/2* and *mul2/2* kernels alternately, in a round-robin fashion. Each parallel branch only processes half the elements, thus effectively doubling the throughput.

One of the main goals of our research is to bring the benefits of FPGA hardware to the software programmers, while hiding the details of the underlying hardware. For example, we plan to hide the details of kernel replication. The programmer only has to identify which kernels are bottlenecks in an application, and then specify how much the throughput of the kernel should be increased. Compiler can then automatically create the necessary replicas of the kernel, or report an error if the kernel cannot be sufficiently replicated due to limited hardware availability. In the current version of our compiler we do not yet support automatic kernel replication. However, we have performed experiments with kernel replication, where we replicated kernels manually,

in a manner that we can envision a compiler could easily perform automatically. We present the results of these and other experiments next.

4. Experimental Evaluation

To validate the correctness of our design flow and estimate the performance benefits of our approach, we implemented two small applications using our flow. We then compared the throughput of our implementation in hardware with the throughput of the best software implementation, running on a Nios II soft processor on the same FPGA device, and the same software function accelerated using C2H. This comparison is fair, because our design flow presents the programmer with a compiler interface that is similar to the traditional software development flow, much like C2H. Comparing our approach to a hard-core processor or a GPU would not be fair, because of significantly differing technologies used for their implementation. Research in [5] and [6] has shown that streaming programs can be efficiently compiled and executed on general-purpose processors and GPUs, respectively.

We chose two applications that are often used to demonstrate computation acceleration, because they are simple and their characteristics are well understood. These applications are Autocorrelation and Finite Impulse Response (FIR) filter. Autocorrelation is an operation that computes the cross-correlation of a signal and a shifted version of that same signal. In our experiments we perform autocorrelation of a signal consisting of 100,000 samples for 8 different shift distances. The FIR filter is commonly used in digital signal processing as a digital filter. The filter works by storing a certain number of samples in a pipeline and then multiplying each sample by a constant factor and summing those products. The depth of the pipeline is often referred to as the number of taps in the filter. In our experiments we use a filter with 8 taps on an input signal consisting of 100,000 samples. In both applications samples were represented as 32-bit integers. In all our experiments we use FIFOs with depth 4. We found that increasing the FIFO sizes beyond 4 was not beneficial for our benchmarks.

Our experimental system is based on the Nios II processor f(ast) version, with instruction and data caches (4 KBytes each), and a hardware multiplier unit. The processor is connected to an off-chip 8-MB SDRAM module, and the *timer* and *UART* peripherals which enable measuring and reporting the program execution times. Software implementations of both applications were first run on this system and their throughput was recorded, along with the area and the maximum operating frequency (F_{max}) of the system. Next, we implemented each application in the Brook streaming language, and compiled it using our basic flow, with each kernel mapped to one hardware accelerator. We measured the area and throughput of each application, and

then replicated the kernels in each application 2 and 4 times. Finally, we accelerated the original software function using C2H. All experiments were run on Altera’s DE2 development board, with a Cyclone II EP2C35F672C6 FPGA device. Software was run from an off-chip SDRAM memory, because the large dataset of 100,000 elements could not fit into the available on-chip memory.

4.1. Results

Results of our experiments are summarized in Table 1. The first column indicates the application, where *autocor_soft* and *fir_soft* correspond to software implementations, while *autocor_c2h* and *fir_c2h* correspond to C2H implementations. *autocor* and *fir* correspond to the basic streaming implementations, whereas the applications with *x2* and *x4* in their name correspond to the applications whose kernels were replicated 2 and 4 times, respectively. The third column presents area results for the logic performing the computation. We do not include peripherals and units that are in the system just for the measurement and debugging purposes in this area, because they are not necessary once a real system is deployed. For software implementation, this means that we only report the area for the Nios II processor and the SDRAM controller. For streaming and C2H implementations we only report the area for the accelerators, FIFOs (where applicable) and the SDRAM controller. We do not include the area for the Nios II processor, because its role of starting the accelerators could easily be replaced by a simple state machine. Last two columns in the table present the system’s F_{max} , and throughput relative to the software implementation.

There are several interesting observations that can be made. First, it is interesting to note that the streaming implementation and the software implementation of autocorrelation achieve similar throughputs. This is because the operations are simple and the input data is generated on-chip, so the processor can fit the loop code into its instruction cache and perform computation without accessing the off-chip memory. The off-chip memory is accessed only when a result is written, which is the same behavior as that of the streaming implementation. As a consequence, the streaming version and the software version exhibit similar performance; the small difference is due to different F_{max} of these two systems. C2H implementation exhibits similar behaviour, but its throughput is lower due to even lower F_{max} . This is because C2H implements complete algorithm in one accelerator, while the streaming approach distributes it across several accelerators.

One significant difference between the streaming implementation and the other two approaches is that the throughput achieved by the streaming implementation can be improved by replicating its kernels. As our results show, replicating the kernels two or four times results in nearly double

Table 1. Throughput and area results for different application implementations

Application	Throughput (KB/s)	Area (LEs)	F_{max} (MHz)	Relative Throughput
autocor_soft	6,255	2,981	142	1
autocor	5,878	2,353	132	0.94
autocor_x2	12,240	3,844	138	1.96
autocor_x4	23,833	6,706	134	3.81
autocor_c2h	5,509	1,359	124	0.88
fir_soft	2,163	2,981	142	1
fir	7,948	3,599	130	3.68
fir_x2	15,515	6,236	127	7.17
fir_x4	19,141	11,505	127	8.85
fir_c2h	5,849	1,775	113	2.70

and quadruple throughput, respectively.

The situation is slightly different for the FIR filter application. In this application the incoming samples have to be inserted into a shift register. In both implementations, this shifting is implemented as a circular buffer in memory. This operation is more efficiently implemented in hardware because independent operations (e.g. updating the loop counters and writing to the buffer) can be performed in parallel. As a result, C2H implementation achieves 2.7 times, and streaming implementation achieves 3.68 times higher throughput than software. Throughput of the streaming application can be additionally increased by replicating the kernels. Doubling the number of kernels results in double throughput, as expected. However, when the kernels are replicated four times, we fail to achieve four times higher throughput, because the implementation of the shift register cannot be easily replicated automatically. The kernel requires all 8 elements of the input to be available to assign them to outputs in a round-robin fashion. Therefore, replicating the node does not reduce the amount of work each node has to perform. Although it is conceivable that this could be improved manually, there does not seem to be an easy way to perform it automatically. Therefore, once the shift register implementation becomes a bottleneck, performance cannot be automatically improved any more.

Comparing the area results, C2H implementations require less area for implementation than equivalent streaming implementation, but they also provide lower F_{max} and throughput. In addition, streaming kernels can be replicated to further increase the throughput, while C2H does not provide such an option.

5. Concluding Remarks

In this paper we presented a novel approach for allowing software programmers to target FPGAs. The streaming paradigm allows programmers to effectively express parallelism, and it maps well to the FPGA logic. We presented a design flow that converts a Brook streaming program into hardware using our source-to-source compiler and Altera's

C2H compiler. Many FPGA systems currently use software running on a soft processor to implement part of functionality, while critical portions of the application are described in an HDL and implemented in hardware. Our system allows the application to be fully described in software and still exploit the capabilities of FPGA hardware, thus reducing design time and cost. Our experiments show that this approach results in up to 8.9 times better throughput than a soft-core processor, and up to 4.3 times better throughput than a C2H accelerated implementation running on the same FPGA. Moreover, the performance can be improved by employing more hardware to perform the computation. Our future work will focus on automating replication of kernels to increase throughput automatically. We also plan to build several large applications to demonstrate usability of our approach for real-world applications.

References

- [1] Altera. Nios II C-to-Hardware Acceleration Compiler, November 2007. <http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html>.
- [2] I. Buck. Brook Spec v0.2, October 2003. Tech. Report CSTR 2003-04 10/31/03 12/5/03, Stanford University.
- [3] D. Tarditi et al. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 325–335, 2006.
- [4] G. DeMichelli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, New York, NY, 1994.
- [5] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *Proc. 38th Int. Symp. on Microarchitecture*, pages 343–354, Washington, DC, 2005.
- [6] I. Buck et al. Brook for GPUs: Stream Computing on Graphics Hardware. *Trans. on Graphics*, 23(3):777–786, 2004.
- [7] L.W. Howes et al. Comparing FPGAs to Graphics Accelerators and the Playstation 2 Using a Unified Source Description. In *Int. Conf. on Field-Programmable Logic*, 2006.
- [8] M.I. Gordon et al. A Stream Compiler for Communication-Exposed Architectures. *ACM SIGOPS OS Review*, 36(5):291–303, 2002.
- [9] A. B. P. Mukherjee, R.; Jones. Handling Data Streams while Compiling C Programs onto Hardware. *Proc. IEEE Computer Society Annual Symp. on VLSI*, pages 271–272, 2004.
- [10] N. Bellas et al. Template-Based Generation of Streaming Accelerators from a High Level Presentation. *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2006.
- [11] S-W. Liao et al. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proc. Int. Symp. on Code Generation and Optimization*, pages 196–207, Washington, DC, 2006.
- [12] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.
- [13] W.J. Dally et al. Merrimac: Supercomputing with Streams. *2003 Conference on Supercomputing*, pages 35–35, 2003.
- [14] Y. L. C. N. W. Wong. Generating hardware from OpenMP programs. *Proc. IEEE Int. Conf. on Field Programmable Technology*, pages 73–80, Dec. 2006.