# AN ARCHITECTURAL FRAMEWORK FOR AUTOMATED STREAMING KERNEL SELECTION

*Nikolaos Bellas, Sek M. Chai, Malcolm Dwyer, Dan Linzmeier*

Embedded System Research Lab
Motorola, Inc.
{nikos.bellas@motorola.com}

## ABSTRACT

Hardware accelerators are increasingly used to extend the computational capabilities of baseline scalar processors to meet the growing performance and power requirements of embedded applications. The challenge to the designer is the extensive human effort required to identify the appropriate kernels to be mapped to gates, and to implement a network of accelerators to execute the kernels. In this paper, we present a methodology to automate the selection of streaming kernels in a reconfigurable platform based on the characteristics of the application. The methodology is based on a flow graph that describes the streaming computations and communications. The flow graph is used to efficiently identify the most profitable subset of streaming kernels that optimize performance without exceeding the available area of the reconfigurable fabric.

## 1. INTRODUCTION

The levels of integration of modern FPGAs have advanced to the point where complex SoCs with processors, accelerator IP, peripherals, and system software can be built and deployed very fast. Tool vendors have offered a plethora of predefined IP cores for frequently used kernels in multimedia, communications, networking, etc. What is missing is a methodology for an application developer to extract computationally complex kernels from the application and map them to gates in an automated way. The availability of a tool flow that abstracts out the hardware details of a module or a set of modules and presents a familiar software-only programming model will be crucial for the acceptance of FPGAs from a large pool of software engineers and algorithm developers.

Central to the design of such a tool is the automated selection of an optimal subset of kernels under area constraints. Reconfigurable logic is customized post-fabrication, and has only finite number of logic cells to implement an application. It is often the case that the hardware designers have to iterate multiple times and perform manual software hardware partition of an application before a successful generation of the FPGA bitstream. This paper describes a methodology for automatic selection of kernels in a streaming application and their mapping into an network of accelerators. The selection is accomplished using intelligent analysis of the computational complexity of the kernels and the flow of the streaming data between the kernels. Kernels are selected to be mapped in gates based not only on their execution time, but also on their data communication profile, and their inherent parallelism and speed-up potential. The most important aspects of the selection process is the efficient representation of the streaming domain and the exploration of the design space without artificially limiting the potential solutions.

Our methodology is using a holistic approach by considering the performance of the whole streaming application. Similar work has focused recently on expanding the processor ISA by identifying application hotspots [1][2]. The main contributions of the papers are the following: first, we describe how a streaming application is represented using an annotated stream flow graph (SFG) and we outline the metrics used for the annotation. Second, we detail the algorithms used to select a near optimal set of kernels to be mapped into gates based on the annotated SFG. Although the focus of the paper are FPGAs, the techniques described can be naturally extended for ASICs.

The rest of the paper is organized as follows: Section 2 gives background information on the streaming programming paradigm, explains the structure and attributes of the SFG, and describes the algorithms used to select the kernels to be mapped in gates under area constraints. Section 3 presents results for a set of streaming applications, and section 4 concludes the paper.

## 2. STREAMING KERNEL SELECTION

### 2.1. Streaming programming model

Programs that follow the streaming paradigm are expressed as an interconnect of filters that communicate using streams. The streaming programming model separates communication from computation, and favors data intensive applications with a regular memory access patterns [3]. Properties of streaming model of computation include [4]:

- Computation kernels are independent and self-contained

Computation kernels are localized such that there are no data dependencies between other kernels. A programmer can

1-4244-0910-1/07/$20.00 ©2007 IEEE

annotate portions of a program that exhibit this behavior for mapping onto a stream processor or accelerator.

- Computation groups are relatively static

The processing performed in each computation group is regular or repetitive, which often come in the form of a loop structure. There are opportunities for compiler optimization to organize the computation as well as the regular access patterns to memory.

- Explicit definition of communication

Computation kernels produce output streams from one or more input streams. The stream and other scalar values which hold persistent application state are identified explicitly as variables in a communication stream or signal between kernels.

- Limited lifetime of the stream data

There is a small amount of reuse for each stream element. Each stream is usually consumed by one or more kernels, which perform little processing on each stream.

In this work, the location and shape of streams in the memory is defined using stream descriptors [5]. The tuple *(Type, Start_Address, Stride, Span, Skip, Size)* can describe a stream with elements of *Type* stored as a 2D array starting at location *Start_Address*, so that:

- *Stride* is the spacing, in number of elements, between two consecutive stream elements.
- *Span* is the number of elements that are gathered before applying the skip offset
- *Skip* is the offset is applied between groups of span elements, after the stride has been applied
- *Size* is the number of elements in the stream

Multidimensional and even non-rectangular stream shapes can be described by extending the tuple definition of the streams. An example of 2D stream is shown in Figure 1.

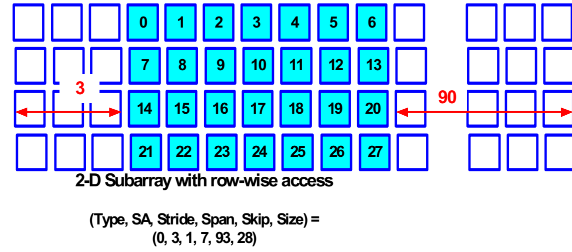



**Figure 1** Stream descriptors for a row-wise memory access pattern

### 2.2. Method Overview

The objective of an automated method for streaming kernel selection is to be used as part of a high level tool that drives the system level architecture of the network of accelerators. The algorithm for kernel selection is shown in Figure 2.

The application is expressed using explicit streaming constructs that identify the computational kernels and the streaming channels used to transfer data. We are using a programming model, that expresses streaming kernels using Data Flow Graphs (DFGs), and streams using stream descriptors [6]. The programmer or a high level compiler analyze the program and identify critical computational kernels in the code that will be executed by a streaming accelerator. The kernels are translated to a machine independent DFG, in which all the data dependencies are explicitly stated in order to facilitate parallel execution.

System level constraints such as maximum available area in number of CLBs or equivalent gates, and available memory and bus bandwidth are given as input to constrain the problem. Finally, profiling data of the execution time of each kernel, and its bandwidth can be optionally used if available. Due to the static and regular nature of computation and communication in the streaming domain, prior runs of the applications may be unnecessary for some applications.

The first step of the kernel selection is to build the SFG data structure, based on the streaming data flow and the available hardware resources that participate in the application. Then, the nodes and edges of the SFG are annotated with metrics that summarize the execution profile of the application, and form the basis for the solution space exploration in the next step. In this work, we describe two strategies to select kernels. The first strategy is to iteratively select streaming kernels based simply on their annotation in the SFG. The second strategy adapts to the current selections that have already been made and continuously changes the annotation of the unselected kernels to better capture the dynamic nature of the selection process. For example, the second strategy favor neighboring kernel nodes of already selected kernels in order to improve the data locality of the communication and avoid costly transfer to the main memory.

A list of kernels sorted with respect to their selection order is produced at the end. One of the strength of the method is that no assumptions is made on the number and type of accelerators used for the low level implementation. For instance, all the selected kernels of the application can be mapped into a single accelerator, or each kernel to a

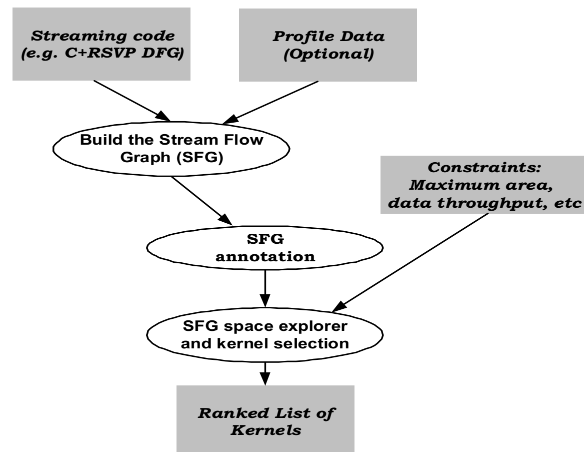



**Figure 2** High level kernel selection diagram

dedicated accelerator, or any hybrid implementation between these two extremes.

### 2.3. Annotated stream flow graph (SFG)

The stream flow graph of an application program *P*in a system *S* is a directed graph *G(P,S) =(V,E)* in which:

- a vertex $u \in V$ can be one of the following types: *kernel* node that expresses streaming computation, *buffer* node that expresses temporary buffers, *main memory* node that expresses off-chip main memory, and *peripheral* node that expresses peripherals that source or sink streams (e.g. image sensors or LCD displays).
- an edge $e = (u,v) \in E$ connects two nodes if there is a stream produced by *u* and consumed by *v*.

The SFG depends on the application and the architecture of the system. The application determines the structure of the SFG, while the system determines the type of nodes that are available and how they can be used.

The SFG expresses static, as opposed to dynamic, stream flow. There is an edge between two nodes *u* and *v* if there is a stream flow between them, and also a thread of control in the code in which first *u* and then *v* is executed (or accessed), even if that thread is not executed in the dynamic program. For instance, in case of a conditional if-then-else or case statement, there will be edges between all potential paths between kernels.

The SFG is built as a preprocessing step during compilation time. If the programmer or an optimizing compiler uses loop tiling to partition the kernel execution across data tiles and to place the communicating streams in tile buffers, the SFG preprocessor instantiates buffer nodes. Otherwise, it instantiates main memory nodes. The preprocessing step of Figure 2 can be used after a source-level optimizing compiler that performs tiling but it does not perform any source code optimizations by itself.

Figure 3 shows the SFG for a tiled implementation of an image processing chain used for processing Bayer color data produced by an image sensor. Tiled frame data are processed by computation kernels one tile at a time. If the programmer did not use tiling, the SFG would contain main memory nodes in between the kernels.

The annotation of SFG nodes is used to capture dynamic execution activity, when the application is driven by a set of input data. Each kernel node $u \in V$ is assigned a value using the guide function $f(u)$, and a cost $c(u)$. The purpose of the SFG annotation is to intelligently rank the kernel nodes so that the best candidates are used for hardware implementation. The guide function is a weighted sum of three metrics that are used to grade the

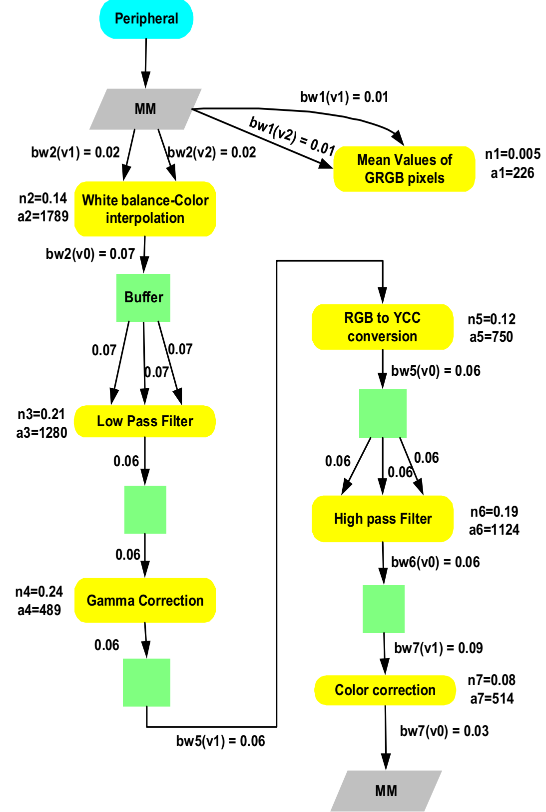



**Figure 3** The Stream Flow Graph for the tiled version of the Image Processing Chain benchmark. Each node *u* is annotated with the computational metric *n(u)*, the parallelism metric *p(u)* (not shown in the Figure), and the area cost *a(u).* Each edge is annotated with the bandwidth metric *b(u).*

computational complexity, the bandwidth, and the potential for parallelism of the kernel:

$$f(u) = w_1 * n(u) + w_2 * (bw_{in}(u) + bw_{out}(u)) + w_3 * p(u)$$

so that: $w_1 + w_2 + w_3 = 1$

The metrics are determined by profiling data or, in some cases, by static analysis of the application code. Different weights $w_i$ will affect the types of candidates selected. The rest of this section details how the parameters of the guide functions are evaluated and what are the trade-offs.

The computational metric $n(u)$ is the execution time of kernel *u* as a percentage of the sum of execution times of all kernel nodes in *V*. The metric assumes a perfect memory system, and it represents only the percentage of computation time, and not of memory accesses overhead. For instance, the Low Pass Filter kernel accounts for 21% of the execution time of all streaming kernels in Figure 3.

The bandwidth metric $b(e)$ of edge *e* equals the number of bytes that were transferred via edge *e* as a percentage of all bytes transferred between all edges in SFG. For a node *u*, $bw_{in}(u) = \sum_{inedges} b(e)$, and $bw_{out}(u) = \sum_{outedges} b(e)$. For the

Low Pass Filter kernel $b_{win}(u)$ = 0.7*3 = 0.21 and $b_{wout}(u)$ = 0.06

The purpose of this metric is to include kernels that process large amount of streaming data. By selecting them, the algorithm can form clusters of high bandwidth kernels so that the data are not transferred back and forth between the accelerators and the memory. We will come back to this observation in the following section.

The metric $p(u)$ considers the complexity of the memory access pattern of node $u$ to evaluate the potential for speed up when $u$ is mapped to gates. The largest performance gains are possible when the streams in and out of the kernel have regular access patterns similar in shape to the order with which data are stored in the main memory (e.g. row-wise). Memory-bound kernels are restricted by the memory access inefficiencies even if a large number of functional units are used to implement the computations. For our methodology:

$$p(u) = \frac{\sum_{\forall s \in S} SAE(S)}{|S| + 1}$$

in which S is the set of all the streams consumed and produced by $u$, and $SAE(s)$, or stream access efficiency, is the number of stream elements of stream $s$ fetched in every bus cycle, on average. Kernels with a large number of I/O streams, and low stream access efficiency, are less likely to be selected. When a kernel is used in multiple locations in the application (potentially with different stream descriptors), the algorithm uses a weighted average value of the SAE values.

As an example, consider the simple vector add DFG kernel of Figure 4. Assuming that the system bus can fetch 8 bytes per cycle, the *SAE* values are:

$SAE(v1) = 4/8 = 0.5,$
$SAE(v2) = 1/8 = 0.125,$
$SAE(v0) = 8/8 = 1$
$p(u) = (0.5 + 0.125 + 1) / 4 = 0.28$

The cost of selecting a node $u$ is equal to the area complexity of the node $a(u)$. Since the area of the accelerator implementation is unknown before scheduling and binding is performed, the algorithm uses an area estimation metric that is proportional to the number, type, and bitwidth of the nodes of the DFG of node $u$. To that effect, a predefined hardware table is used that stores the area cost of each node type of the DFG. The hardware table was estimated using implementations of functional units in a Xilinx Virtex-4 FPGA. This cost is scaled to match the bitwidth of the specific node. The hardware table considers the area complexity of computational nodes, and of stream push and pop nodes. These nodes create streaming units that are separate from the data path

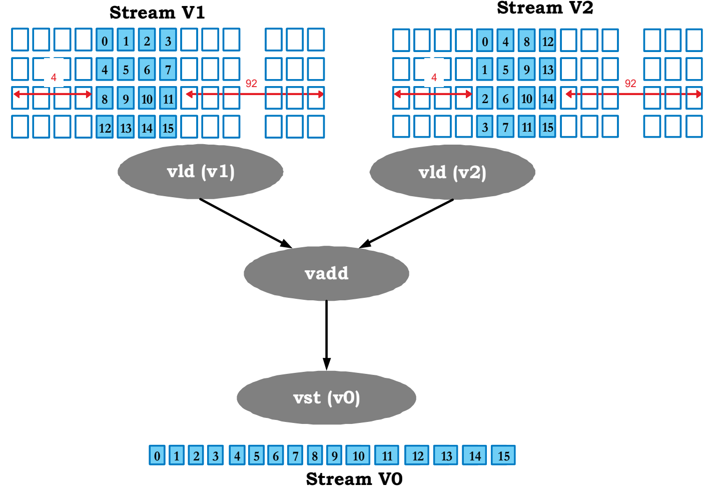



**Figure 4** A simple vector add DFG

but contribute substantially to the final area.

Although the area of the accelerator that will finally implement the node $u$ is probably different than what this method computes, what is important in this step is the consistency of the area estimation. In other words, a more complex kernel with a higher cost $a(u)$ should also be implemented in a larger accelerator. More details on the methodology of pre-synthesis area estimation can be found in [5].

The weights $w_i$ are user defined. The weight $w_2$ is equal to zero for SFG edges that correspond to a transfer of streaming data between a kernel and the main memory. In that case, selecting neighboring kernels does not offer any advantages because the streams will be stored to main memory, and temporary storage is not possible.

### 2.4. SFG space exploration and kernel selection

Based on the SFG formulation, the next step is the selection of an optimal set of kernels that maximizes the value $V$ under an area constraint $A$. The selection is similar to the 0/1 knapsack problem, which is NP-complete. Given a set of resources (the kernel nodes), with each resource having a value $f(u)$ and cost $a(u)$, the objective is to maximize the value of selected resources for given maximum area $A$. The problem can be solved optimally in pseudo-polynomial time using dynamic algorithms. As the experiments show, a simple greedy algorithm works almost as well as the dynamic algorithm shown in Figure 5. In the greedy algorithm, the next kernel $u$ with the highest $\frac{f(u)}{a(u)}$ is selected. In dynamic algorithm of Figure 5, the DYN_COST_1 procedure is called first to compute the value array, in which the entry $C[i][a]$ contains the maximum value when only $i$ kernels are present, and the

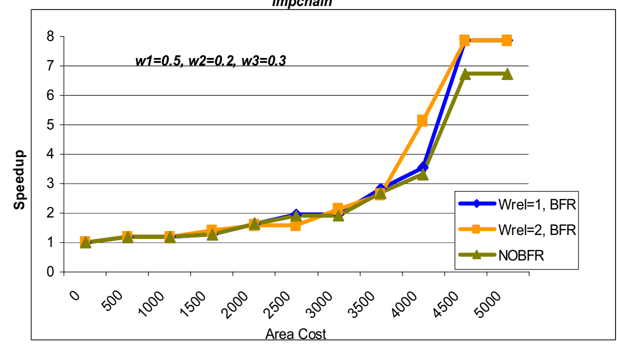
```
DYN_COST_1
Input: f[0..N-1], a[0..A-1], N, A;
Output: C[0..N-1];
{
    C[0, 0..A] = 0;
    C[0..N, 0] = 0;
    for (i = 0; i ≤ N; i++) {
      for (a = 1; a ≤ A; a++) {
        if (a_i > a)
          C[i, a] = C[i-1, a]
        else
          C[i, a] = max{C[i-1, a], f(u_i) + C[i-1, a- a_i]}
        }
  }
  return C;
}

DYN_SEL_1
Input: C[0..N][0..A], a[0..A-1], v[0..N-1], N, A
Output: ranked nodes R
{
      i = N; j = A;
     R={};
      while (i > 0 && j > 0) {
        tmp = a[i-1];
        if (C[i-1][j] ≥ f[i-1] + C[i-1][j-tmp])
           i = i-1
        else {
          if (i > 0 && j > tmp) {
            R = R ∪ u_i
            j = j – tmp;
          }
          i = i-1;
        }
      }
      return R;
}
```

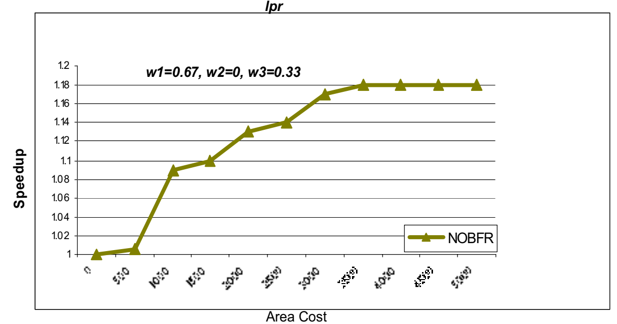
**Figure 5** Dynamic algorithm for kernel selection

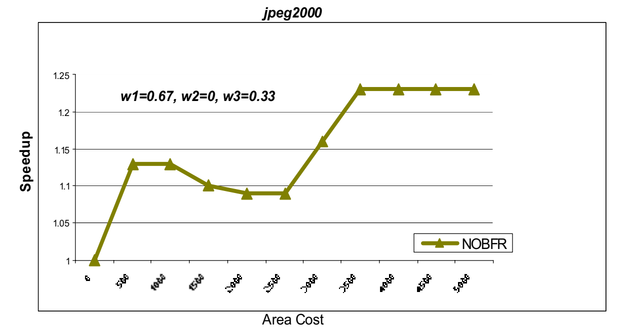
maximum area is *a.* Then, DYN_SEL_1 traverses the array *C* to select the set of kernels. Our approach is extended to adapt to the dynamic flow by favoring kernel nodes that are adjacent to already selected nodes. Once a kernel node *u* is selected, the value *f(v)* of all nodes *v* that are connected with *u* via a buffer node is scaled up by a user defined factor $w_{rel}$. This dynamic update facilitates the clustering of nodes so that streaming data do not need to be accessed from memory unnecessarily. The dynamic programming heuristic generally does better than greedy approaches but the difference is small especially if the problem size is small, i.e. there is a small number of kernels in the application.

## 3. EXPERIMENTAL EVALUATION

The proposed system was built as part of a streaming compiler infrastructure [7]. The kernel selection algorithms were implemented as a separate module from the main compiler, simulator and profiler. We used several streaming applications written for the RSVP™ accelerator to evaluate the kernel selection methodology. The benchmarks were an image processing application (*impchain*) used to perform a sequence of color processing and color conversion filters on a image sensor input frame (Figure 3), a license plate recognition application used to identify vehicles based on their license



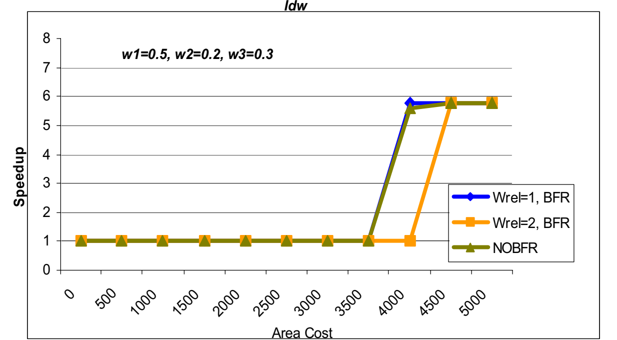




**Figure 6** Speedup for various benchmarks. The NOBFR legend means that the streaming application was written without temporary storage fo streams between kernels (i.e. the streams are always read and written from/to the main memory. The rest of the curves assume intermediate buffering of at least one of the streams.

plates, a *JPEG2000* image compression/decompression program, and an automotive lane departure warning application used to detect road lanes for driver assistance. We are selecting applications with multiple streaming kernels with a complex streaming flow to better illustrate the feasibility of the approach. We used profiling and static analysis on each of the applications to determine the value and cost of each streaming kernel. The area cost estimates in the hardware library were calculated by implementing and synthesizing every DFG node, as explained in section 2.3. The area cost of a kernel is approximated as the sum of the costs of all the nodes of that kernel.

The baseline machine for the experiments is an ARM946 RISC processor, and the speedup ratios of Figure 6 are expressed with respect to the baseline performance as the area cost varies. Each line in an application represents the speedup of the application compared to the baseline machine for a specific set of selection criteria (values of weights $w_i$). We experimented with various combinations of weight values to determine if there were combinations that consistently resulted in higher speed up at each area cost point. The experimental analysis showed that the weight combinations were slightly different for each benchmark. However, the $n(u)$ metric was consistently weighed more for higher speed ups, as shown in the results of Figure 6.

One of the interesting observations is that the speedup varies a lot across benchmarks. The *impchain* and *ldw* benefit a lot from hardware acceleration because almost all of their computation is a series of kernel filters. The other benchmarks have a large portion of the program being spent on branches and pointer operations that hinder mapping on streaming computations.

The dynamic update of kernel values was used only in the *impchain* and *ldw* benchmarks because these application are using tiling ($w_2 > 0$). The three curves in each benchmark correspond to different values of the weights:

a) $w_1 = 0.5$, $w_2 = 0.2$, $w_3 = 0.3$, $w_{rel} = 1$,

b) $w_1 = 0.5$, $w_2 = 0.2$, $w_3 = 0.3$, $w_{rel} = 2$ and,

c) $w_1 = 0.5$, $w_2 = 0$, $w_3 = 0.3$, $w_{rel} = 1$.

When $w_{rel} > 1$, the dynamic kernel update props up the value of all the neighbours of a selected kernel by a factor of $w_{rel}$.

For the *lpr* and the *JPEG2000*, we set $w_2 = 0$, $w_{rel} = 1$, because all the streams in between kernels are spilled to the main memory, and there are no buffer nodes between kernels (NOBFR). The dynamic update of the node values does not always result in a better solution for a given area limit, because it may favor kernels that contribute less in the total execution time even if they are neighbors of already selected nodes.

In the experimental evaluation of Figure 6, every kernel selection includes all appearances of the kernel in the application. There are cases where the kernel hardware

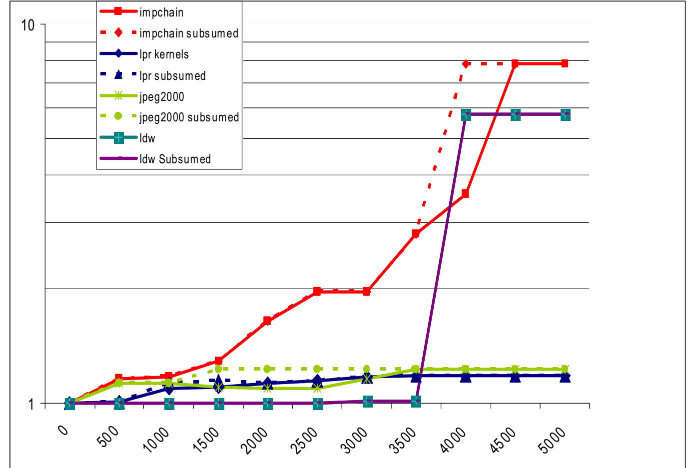



**Figure 7** Speedup due to subsumed kernels (logarithmic scale)

can be generalized to execute more than one kernel with little or no extra area cost. For example, an accelerator that computes the dot product of two complex vectors can be used to compute the sum of two integer vectors. The continuous lines of Figure 7 show the speedup when no generalization is supported, and the dashed lines show the speedup when the hardware is extended to support the execution of a similar but no larger kernel that has not yet been selected.

The experiment shows that hardware generalization is a very useful mechanism in some cases. For example, almost all the DFGs of the *JPEG2000* benchmark are similar, and can be mapped to the same hardware without any extra area penalty. The *impchain* and *ldw* benchmarks, on the other hand, consist of large kernels with limited commonality. Using graph generalization is particularly important in cases of limited area budget.

## 4. CONCLUSION

Hardware accelerators customized for a particular task and implemented in hardware are an efficient way to enhance system performance and meet application requirements. This paper presents a methodology to automate the selection of streaming kernels that are mapped in hardware accelerators in a reconfigurable fabric. The methodology is flexible and can be tuned by the user to match the application and the targeted device characteristics. It exploits the parallelism inherent in a lot of applications and has demonstrated that a small amount of extra fabric area can result into significant performance gains. In the future, we plan to integrate this tool to a larger architectural synthesis program that automates the generation of hardware given a high level representation of an application.

## 5. REFERENCES

[1] Fei Sun, Ravi, S., Raghunathan, A., N.K. Jha, "Synthesis of custom processors based on extensible platforms," *Proceedings of the International Conference on Computer Aided Design (ICCAD),* November 2002, pp. 641-648

[2] Nathan Clark, Hongtao Zhong, Scott Mahlke, "Processor Accelerator Through Automated Instruction Set Customization," *Proceedings of the 36th International Conference on Microarchitecture*, December 2003

[3] Amarasinghe S., Thies B. "Architectures, Languages and Compilers for the Streaming Domain" in *tutorial at the 12th Annual International Conference on Parallel Architectures and Compilation Techniques,* New Orleans, LA

[4] Sek M. Chai, Nikolaos Bellas, Malcolm, Dwyer, Dan Linzmeier. "Stream Memory Subsystem in Reconfigurable Platforms", *in the 2nd Workshop on Architecture Research using FPGA Platforms (WARFP).* February 2006, Austin, TX

[5] Somsubhra Mondal, Seda O. Memik, Nikolaos Bellas. Pre-synthesis area estimation of reconfigurable streaming accelerators. 16th International Conference on Field Programmable Logic and Applications (FPL), August 28-30 2006, Madrid, Spain

[6] Chirisescu S., et. al. " The Reconfigurable Streaming Vector Processor, RSVP™" *in Proceedings of the 36th International Conference on Microarchitecture,* December 2003, pp. 141-150, San Diego, CA

[7] Nikolaos Bellas, Sek M. Chai, Malcolm Dwyer, Dan Linzmeier. "FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators" in the 13th *Reconfigurable Architectures Workshop (RAW),* April 2006, Rhodes, Greece