Overview of Stream Processing

Lecture #16: Tuesday, 28 May 2002
Lecturer: Prof. Bill Dally
Scribe: Jacob Chang, Njuguna Njoroge
Reviewer: Mattan Erez

Overview of Stream Processing

- Concept
- Motivation
- Hardware
- Software

1 Three Concepts in Stream Processing

1.1 Streams

Streams are a sequential accesses to a string of records by the computational machines. They can be stripped into sub-streams for computation.

1.2 Kernels

Kernels are the computational units used on streams. The advantage of thinking in terms of kernels is that their global accesses are made explicit. For example, all data for the kernel in Imagine must be placed as an input or as an output stream in a kernel argument.

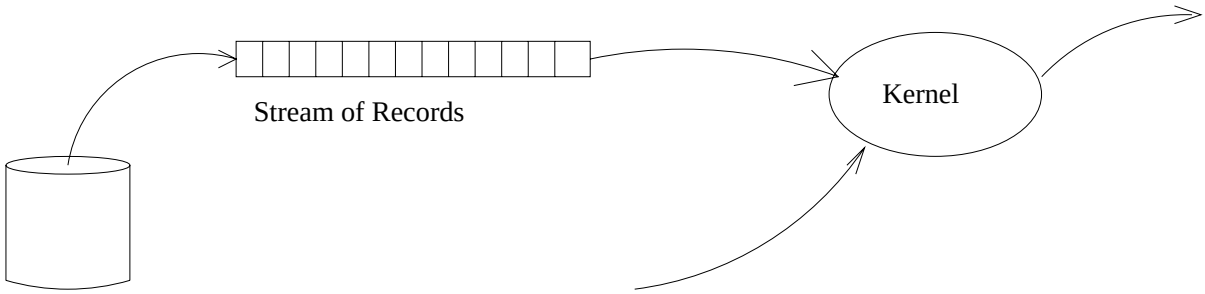

Figure 1: concept of streams

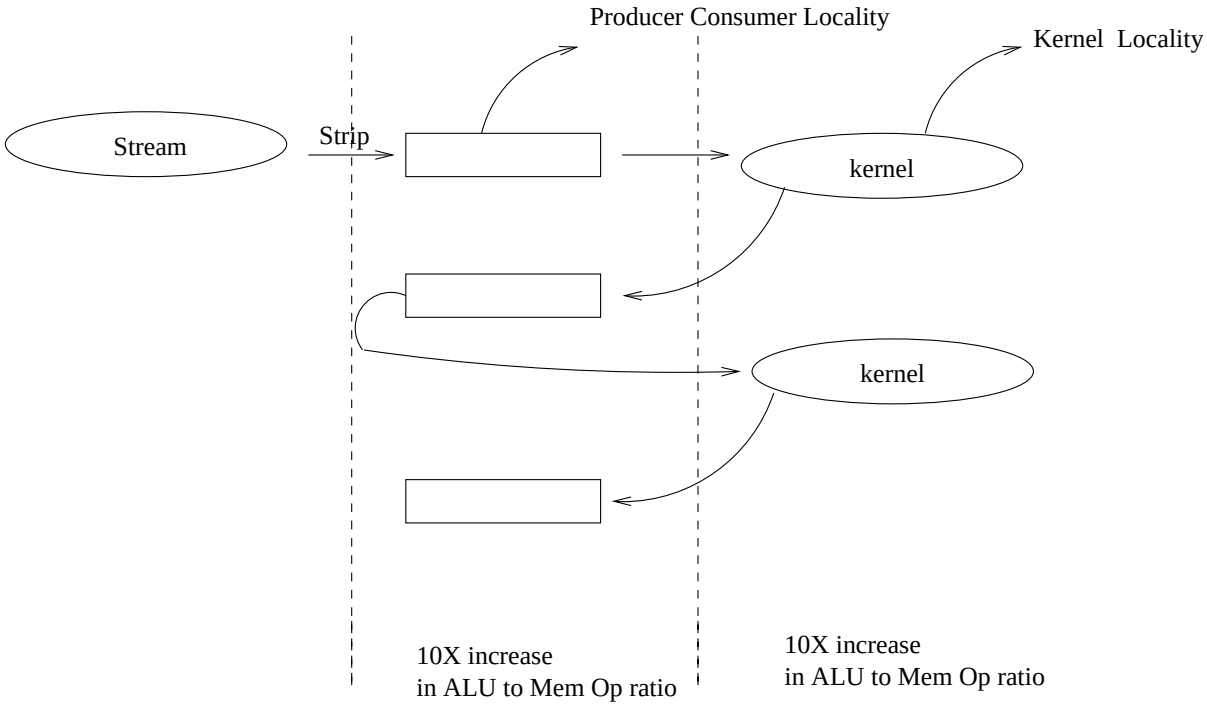

Figure 2: bandwidth hierarchy

1.3 Bandwidth Hierarchy

A traditional processes tries to reduce the off-chip memory access by the use of caches. Caches takes advantage of the spatial and temporal locality. But a stream processor takes advantage of the producer/consumer locality.

The Stream processor utilizes a bandwidth hierarchical system which keeps most of the data movement local (about 90%), some of it spills out to the stream-register files (about 9%) and the remainder rarely accesses the off-chip memory (1%). This system helps hide the latency of the off-chip memory accesses.

A stream processor is particular proficient in keeping the ratio of memory operations to arithmetic operations very low. A traditional accumulator has 1:1 ratio, a scalar processor, 1:4 and the stream processor 1:100.

2 Motivation

There are several key motivations for designing stream processors, the main two being technology-based and application driven.

ALU's are very cheap and take little space–a 64 bit FPU is smaller than $1mm^2$. What is expensive is the bandwidth (i.e. connecting these ALU's). Each ALU requires control

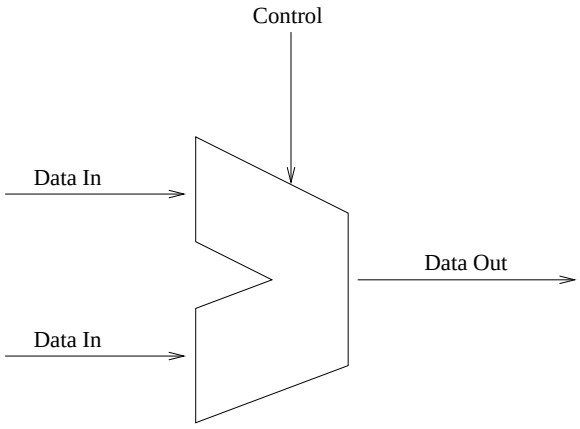

Figure 3: alu connections

data, input data and output data buses.

Furthermore, bandwidth does not scale at the same rate as the growth of ALU units. Therefore, we want our connections to be as close as possible. This has a couple ramifications. First, we reduce demand on BW and second, we only use it where it is inexpensive. Futhermore, we can hide bandwidth latency, which minimizes performance cost.

There is a class of applications that lend themselves to being streamed processed. They have several defining characteristics. First, they provide an opportunity to exploit parallelism. Secondly, they have little or no data re-use, which makes using a conventional cache expensive. Finally, these applications have high computational intensity, which enables a high ratio of computation to data.

3 Stream Hardware

3.1 Definition

Before exploring the hardware of stream processors, we must first come up with a comprehensive definition for a stream processor.

The first criterion for a SP is that it must be able to support streams and kernels. (Note that any current computer would satisfy this criteria.)

Second, a SP exploits kernel locality and producer-consumer locality. Kernel locality refers to granularity on the intra-kernel level. For instance, the output of one ALU can be fed to a neighboring ALU. Producer-consumer locality is on the inter-kernel level, where the stream produced from one kernel is fed into another kernel.

Third, a SP has high arithmetic intensity, which means that it has a high arithmetic computation/bandwidth ratio.

Stream processing hardware can come in several forms. It can be time multiplexed (like Imagine), space multiplexed (like RAW) or a combination of both.

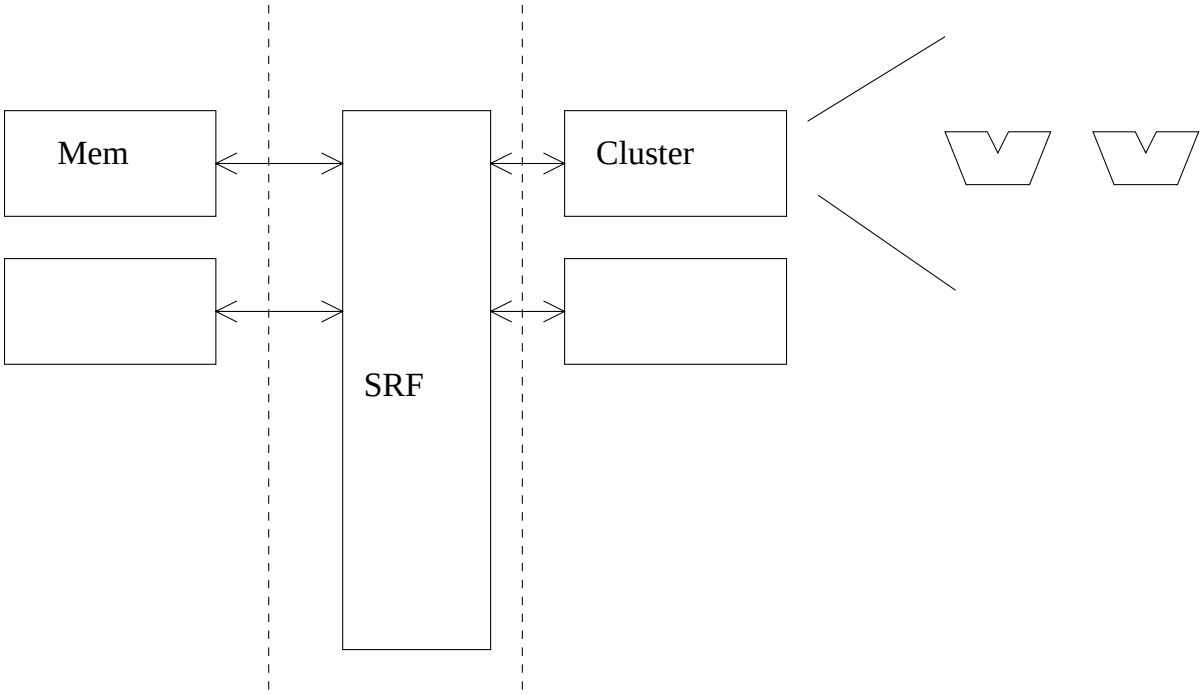

Figure 4: imagine style processor

3.2 Imagine Style Stream Processor

As illustrated in the diagram 3.2, the SP is connected to off-chip memory, data from which is fed into the on-chip Stream Register File. The data from the SRF is fed into stream buffers, which eventually feed the 8 clusters. Each cluster has 6 ALU units, each with its own register file.

3.3 Issues for Imagine Design

3.3.1 Register Organization

There are different ways to organize a processor's register file. The traditional implementation is a centralized register file. This does not work too well with large numbers of ALU units because the size of the RF grows N^3 as the number of ALU increases. A practical alternative is use a distributed register file as shown in figure 3.3.1. A DRF size grows with N^2 as the number of ALU increase. Clustering (making it SIMD) a centralized architecture reduces the area usage by a constant factor (divides the area by the number of clusters). Applying clustering on a DRF also reduces the area by the number of clusters. Finally, applying a memory split leads to the best area usage. You split the register file into two main partitions: a sector with low capacity, but large number of

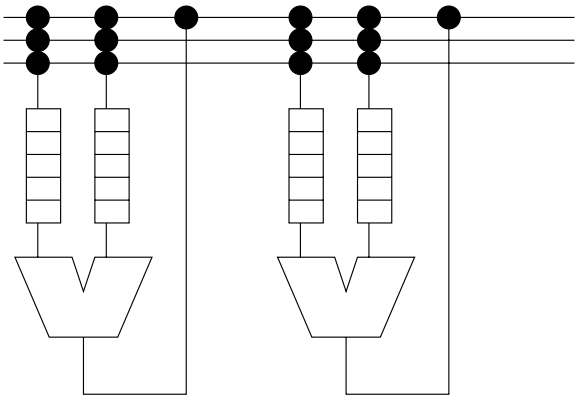

Figure 5: distributed register file

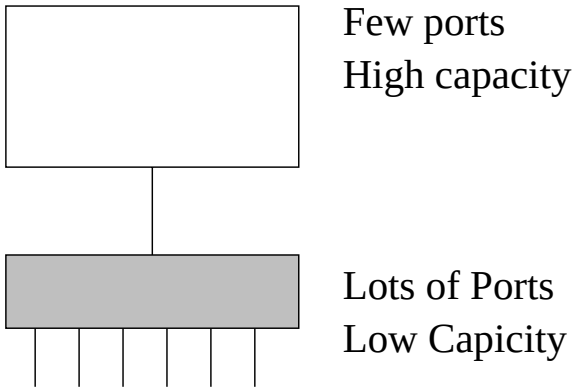

Figure 6: split register file

ports (high BW) and another sector with higher capacity, but small number of ports (low BW) as shown in figure 3.3.1. Figure 3.3.1 illustrates the performance of the various register file types.

Stream buffers act as the intermediary between the SRF and the clusters' register files in a stream processor and also between the SRF and the memory system. Stream buffers effectively time-multiplex the single physical port of the SRF into many logical ports that can be accessed simultaneously. Furthermore, stream buffers lead to an area and power efficient implementation that provides a simple, extensible mechanism by which clients can access the SRF.

3.3.2 Control

One can view the programming of Imagine at two levels. One writes a *Stream Program* to deal with the manipulation of Streams such as load and store streams into the SRF. The *Kernel Program* are the instructions to the Clusters. This type of programming

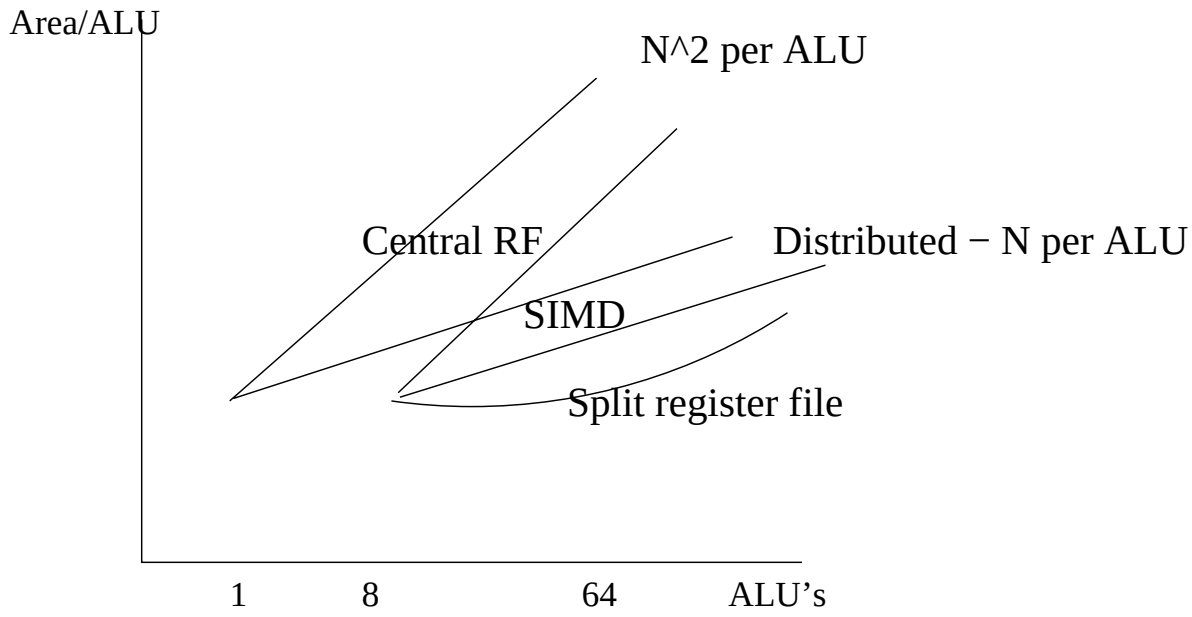

Figure 7: register organization

model localizes the computational intensive units to computational intensive works. The disadvantage of this is that it makes Imagine harder to program.

3.3.3 Conditionals

Conditional statements in programs are typically hard to deal with in a SIMD architecture. There are two ways Imagine deals with conditionals inside the program.

One way is to do predication, which is what a typical SIMD machine deals with conditional statements. Predication means that if it encounters a branch, it does both branches on all data. This results in doing unnecessary work, and is especially wasteful for branches that are large. In Imagine predication is performed in software with the `select` operation.

Another way that Imagine can deal with imagine is with conditional streams. This is illustrated in diagram 3.3.3. This method does not waste computations on unnecessary work, but it does have a constant communication cost for communication the conditional outputs and inputs between clusters.

3.3.4 Extending to Multi-node

One can put together multiple Imagines in a networked fashion for computation also. This allows the ability to multiplex the different kernels in space in additional to time. Or it can also be configured to run parallel kernels to exploit data parallelism.

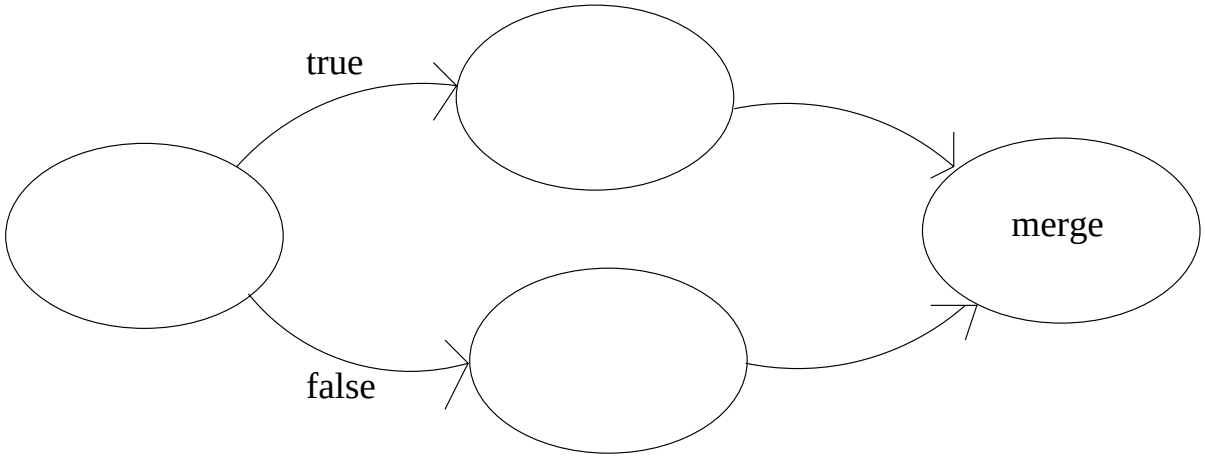

Figure 8: conditional streams

3.3.5 Aspect Ratio

We want to evaluate how "good" an architectural design is based on the performance/cost ratio. The performance aspect of the design not only depends on how many functional units are available on the chip, but also how often they are doing useful work. How many functional units a chip can have is usually not an issue at this time, but how to keep those functional units busy is the real issue.

So the real question is how much bandwidth can we supply to the functional units to keep them busy. Ideally, we would like to keep all the connections local to minimize cost instead of needing to communicate across chip or off chip.

With this in mind, the best way to gain this performance vs. the cost is by exploiting data parallelism of computation. It does not increase the instruction bandwidth, and it allow the results to stay local. In Imagine, this is exploited by having multiple clusters. However, as we increase the data parallelism in Imagine, we will start to run into the short stream effect. When there are short streams, there isn't enough data to keep the cluster busy all the time.

So the next step is to take advantage of instruction level parallelism. Imagine does this by keeping intermediate stream results between kernels in the Stream Register File(SRF) and individual calculations within the kernel in the Local Register File (LRF). However, increasing the ILP will have a corresponding increase in communication cost.

Lastly, one can take advantage of thread level parallelism, where multiple threads can be run at the same time on parallel Imagines. It is arguable that the increasing the ILP will give better tradeoff than increasing TP. But it is believed that ILP does offer a better tradeoff because one does not have to duplicate a lot of the hardware such as the sequencer in ILP.

4 Stream Software

The goal of looking at stream software issue is to easily and efficiently map applications to the stream models. To do this, we will have to look at the type of applications we are trying to map, the languages that can be used to describing a streaming program, and a compiler that is able to may to the hardware efficiently.

4.1 Languages

The goal of stream programming language is to give the user the ability to express parallelism and locality.

One such example is the Stream C/Kernel C language. It is targeted towards the specific architecture of Imagine. It exposes the explicit communication of the Imagine chip and kernel C can only deal with stream accesses.

The StreaMIT language derives the communication implicitly using the peek operation in the language.

Brook uses Stencils to specify the communication pattern.

4.2 Compilation

The goal of the compilation process is to map the application to the underlying hardware.

On the stream level, the compiler needs to deal with the stream manipulations to map the application efficiently. One issue that needs to be dealt with on Imagine is strip-mining. We would like the compiler to strip mine the input data when the input dataset is too large. The other thing we would like the compiler to deal with is software pipelining over the strips. This is so that one can read and execute operations at the same time to hide memory latency. For example, for the operation shown in figure 4.2, one can set the schedule as follows:

Memory Schedule	Kernel Schedule
M1	K0
M2	K1
	K2

So as we can see, the memory operation 1 can be conducted in parallel with kernel 0, and when kernel 0 finishes, hopefully the memory operation will be done and kernel 1 can be started right away.

On the kernel level, we would like to map the kernel to the hardware, dealing with the issue of having multiple, distributed ALUs. The compiler should also deal with ordering the operations and scheduling them as well scheduling the communications.

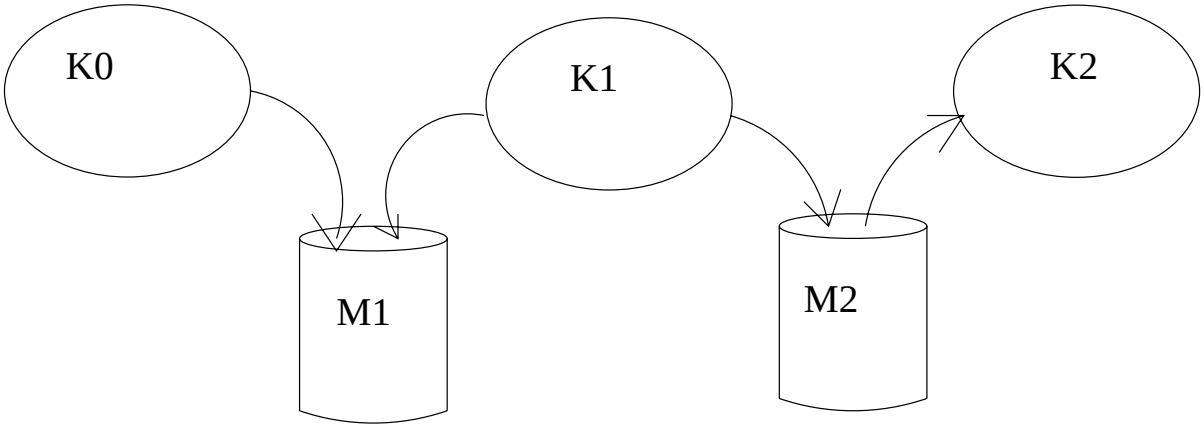

Figure 9: software pipeline example

5 Summary

The Imagine processor have demonstrated to have a 1:100 memory operation to ALU operation ratio, which makes it effective for "media" processing. It can also have the potential for scientific computing study. However, there are issues with writing the software on Imagine. Some things that we would like to still address is to automate the inter-cluster communication, strip-mining, pipeline and scheduling of programs. Basically there is a pressing need to reduce programming complexity on Imagine. Also, other issues to address include figuring out how to convert existing software to run on Imagine, and on the hardware side, find an effectively way to deal with conditionals and explore the use of caches in the processor.