Stream Processor - Continued

Lecture #3: Thursday, 11th April 2002
Lecturer: Bill Dally
Scribe: Nuwan Jayasena, Hsiao-Heng Lee, Francois Labonte
Reviewer: Mattan Erez

During this lecture Some questions were answered by Prof. Dally on stream processing in general and the last two papers reviewed at the last meeting, the imagine general paper and the polygon rendering paper. Then Prof. Dally completed the presentation of the lecture notes from lecture 1. At the end, Mattan presented a quick demonstration of the imagine programming system using the example of comlex number multiplication.

1 Questions and Answer

1.0.1 What happens if some clusters are not fully utilized (i.e. idle)?

We can use conditional streams to balance the load. By adding some small code to check whether clusters are idle every T time, we can load the next stream records in idle clusters. This is enabled by conditionnal streams which will be studied later on.

The granularity of this ”cluster checking” is determined by the programmer. The trade-off here is to spend additional time on checking the clusters to reduce cluster idle time, and checking too often may increase the overhead. Right now the checking must be done manually by the programmer, but it would be interesting if the compiler can automatically provide this functionality.

So the average kernel execution time is approximately ”t3 + floor(t3/t2) * t1” where t1 is the time to check whether the clusters are idle, t2 is the time between each checking and t3 is the total execution time of a kernel.

1.0.2 Scaling of process technology: As the technology scales (larger SRF, more ALUs ...), is compiler scheduling becoming harder?

There are 3 axes of scaling:

- Instruction level parallelism (ILP): more ALUs per cluster
- Data level parallelism (DLP): more clusters and larger SRF bandwidth
- Thread level parallelism (TLP): multiple microprocessors

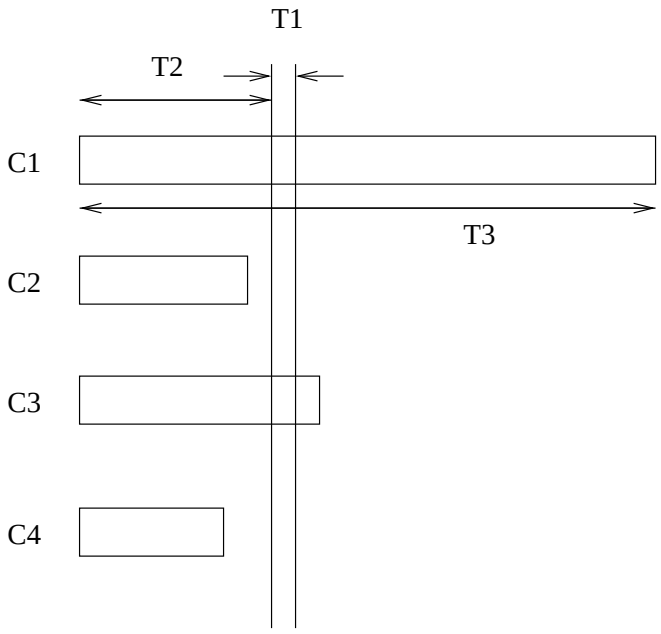

Figure 1: Conditional Streams load balancing

1.0.3 How does the SRF scale?

As the number of clusters grows, the SRF BW demand grows linearly.

1.0.4 How can we schedule threads?

Threads are ”self-scheduling” - not statically scheduled.

1.0.5 Do we need to scale the number of inter-cluster connections as we scale the number of clusters?

No, a study performed by Brucek Khailany (a PhD student in Prof. Dally’s group) has shown that the number of inter-cluster connections remain constant.

1.0.6 Short stream effects

When the streams become too short (< 60 stream elements), the long latency in functional units cannot be amortized and the setup and teardown interaction with the host processor dominates.

1.0.7 Teardown/setup

In order to optimize loop operations, we need to decouple the loop iteration from critical paths (PhD thesis from Prof. Monica Lam). As the number of loop iterations increases, the effective

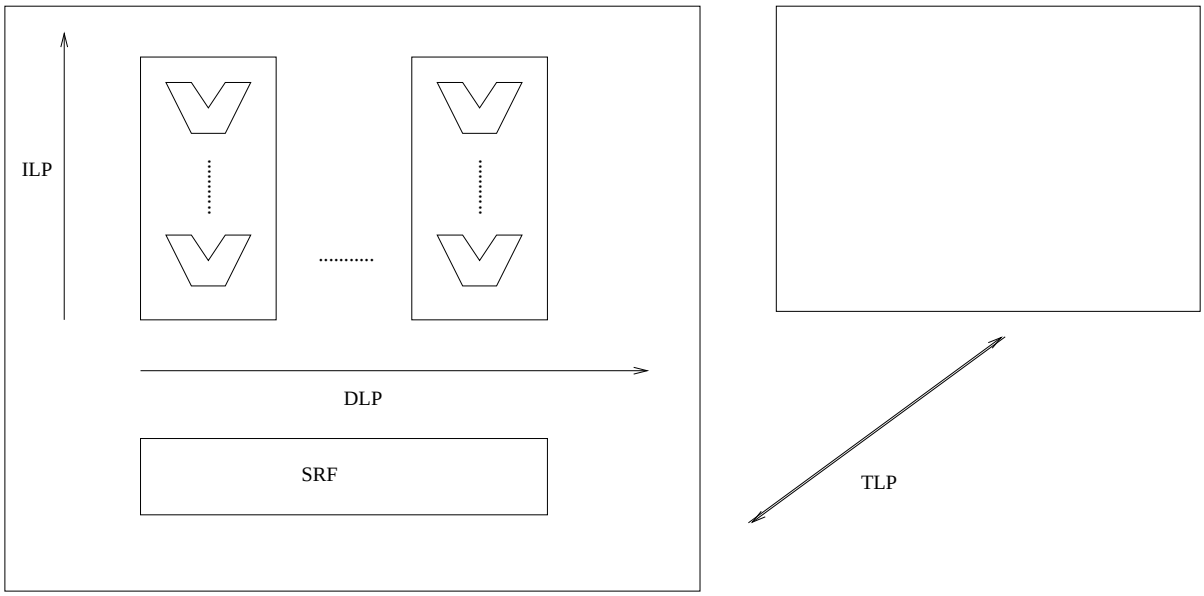

Figure 2: Axes of Scaling

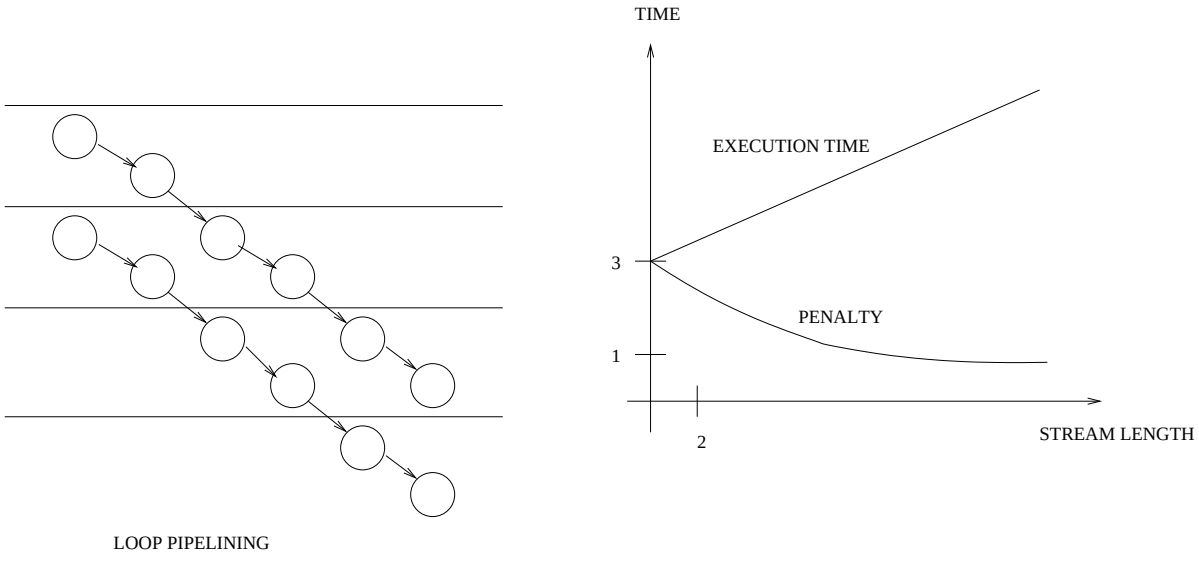

Figure 3: Software Pipeline decouples loop iterations

penalty associated with the loop setup decreases.

1.0.8 Are stream processors programmable?

In the case of Imagine, almost everything is programmable, including ALU communication, intercluster communication, register organization etc. An ALU can communicate with another ALU in one single hop.

1.0.9 What happens if the kernel operations are executed out of order due to variable latency (ie. in the case of polygon rendering, larger triangles need longer time to execute than smaller triangles)?

The results of the data stream on the kernel will come out of order. There are two schemes to solve this problem. One is by indexing each stream element before it enters the kernel then reorder the results based on the indices. In the polygon rendering paper, the triangles need to be properly ordered only if they affect the same pixel. To determine if two fragments generated lie on the same screen x and y coordinates the scratch pad is used in a hash function with the x and y as inputs. Fragments which need to be ordered are put in a stream which is reordered while other common casefragments who don't need orderering just go on.

2 Lecture

(from page 30 of the slides of lecture 1)

2.1 What is a Stream Processor

The stream programming model exposes parallelism, in particular data-level parallelism by doing operations records and across nodes. A stream processor exploits three types of parallelisms:

1. TLP: Thread-level Parallelism across many processors
2. DLP: Data Level Parallelism across many clusters
3. ILP: Instruction Level Parallelism across ALUs inside each cluster

Latency is hidden by performing long pipelined stream loads while computing for a long time a relatively complex kernel (multiple instructions) on a stream of data.

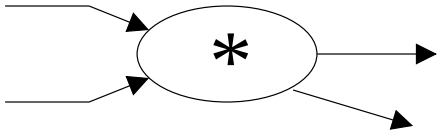

Figure 4: Latency is hidden

A Stream Program also makes data communication explicit and structured. A Stream processor will try to exploit kernel locality within each cluster and the producer-consumer locality within the processor. The bandwidth hierarchy is better exploited by applications which have similar ratio demands on the memory system.

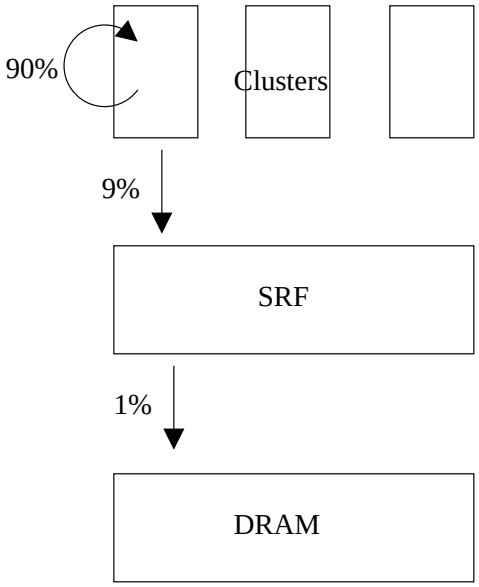

Figure 5: The Communication hierarchy

2.2 The Imagine Stream Processor

Imagine is a load-store architecture for streams where the Stream Register File acts as the Nexus. Streams can be loaded in strides, or indexed from another stream. What the memory system really needs to do is load sequential streams or indexed ones. It is better to use the ALUs of the clusters to generate proper addresses than to make a fancy address generator in the memory controller.

The Host processor runs the StreamC code, loads the KernelC stream programs into the micro-controller memory from the SRF, and decides which KernelC program the clusters will run.

The network interface which sends and receives messages to and from other Imagine processors will move to the Memory System controller in the future Streaming Supercomputer to make network operations like memory accesses.

2.3 Arithmetic Clusters

The ALUs are 32bit FP (can also do integer, 32bit, 16bit and 8 bit operations) they are pipelined with various latency, and can issue one operation per cycle (with the exception of the div/sqrt

unit which is not fully pipelined). The mix chosen is 3 Add/sub, 2 multiply, 1 divide/sqrt. The scratch pad is a 256 word memory that allows to do index memory operation inside a kernel without going back to memory like in the graphics rendering paper where it was used as a hash on screen coordinates to detect conflicts and implement openGL command ordering. The 3 input ports of the scratch pad are write address, read address and write data. It can perform one read and one write per cycle, but no write through (read from address being written). Up to 8 words can be read or written per cycle from the SRF. The clusters communicate with each other through the communication unit one word per cycle.

The registers are located at the inputs of the ALUs, scratch pad and communication unit, all the communication needs to be staged by the compiler through the crosspoint switch. This allows for simple register design, pushing some of the burden on the compiler. There is data replication (duplication) within the registers 2 to 3 times according to Peter Mattson's thesis. The registers can be bypassed such that a result can be fed back to another ALU on the same cycle but if a loop is unrolled enough this is not necessary as latency will be insignificant.

2.4 Bandwidth Hierarchy

The bandwidth hierarchy of Imagine goes from 544GB/s within clusters, 32GB/s to the SRF and 2GB/s to the main memory. Applications don't make full use of all the bandwidths, because of different factors:

1. Overheads in setup and tear down
2. ALUs waiting on memory, memory waiting on ALUs, etc
3. Not perfect parallelism in application
4. Arithmetic operations demand of application differ the ALU mix offering

The ratios of the bandwidth demands is the most important factor to observe. It ranges from 90% to 98% inside the local register file for applications like depth extraction, MPEG encoder, Polygon rendering and QR decomposition. Most important is that the ratio of local register file bandwidth used to memory bandwidth be more than 100:1.

2.5 Physical Implementation

On the die plot, the 8 ALU clusters can be clearly seen on the bottom right, the microcode memory just on top, the SRF and its buffers are left of the clusters. The memory controllers are at the far left, on the top left are the host and network interface.

The chip size, 16mm by 16mm is bigger than initially projected. For the M-Machine, the die size had started big and things had to fall off the chip so Imagine started small and grew bigger. The factors that contributed to bloating:

- Changed from full-custom to standard cell. Full custom requires 10x the effort for 2x more speed, 1/2 the area)
- Foundry partner required full-scan (test procedure to scan in/out serially register contents), which made the SRF and cluster registers twice as big.
- Power estimates were too low, chip is serially bonded so big power busses had to be added
- Some Verilog synthesized areas were underestimated (memory controllers go from 64 to 16 status register)

The ASIC process also had to be improved upon, as the routing, area and timing of the clusters was initially poor. Brucek Khailany is publishing a paper on the methodology for which they grouped 1 bit pitches together to achieve a 300MHz datapath TTTT.

2.6 Streaming Supercomputer

The streaming super computer (SSC) idea was born out of discussions between Bill Dally and Pat Hanrahan regarding the inefficiencies of current supercomputers, which are mostly clusters of symmetric multi-processors (SMPs) connected together by fairly inefficient interconnect. The basic idea was to apply the concepts of stream computation to numerical and scientific applications such as solving ordinary differential equations (e.g. protein folding) and partial differential equations (e.g. fluid flow). Initial studies have shown that the memory accesses of these applications map well to the bandwidth hierarchy of streaming architectures. However, since these applications typically use multi-dimensional and/or less regular data structures compared to media apps, one of the major challenges is the book keeping necessary to orchestrate the explicit memory management.

One objective of the streaming super computer project is to leverage a single micro architecture across a range of machine sizes from single-processor palm-tops to multi-processor workstations to multi-cabinet supercomputers. Table 1 summarizes various potential machine sizes and their key parameters. Figure 6 shows the organization of a multi-cabinet SSC. Table 2 summarizes estimated parts costs for a SSC (without I/O).

Machine size	Nodes	# of FPUs	Peak compute	Total Memory
1 processor	1	64	64 GFLOPs	2 GB
1 board	16	1K	1 TFLOP	32 GB
1 cabinet	1K	64K	64 TFLOPs	2 TB
16 cabinets	16K	1024K	1 PFLOP	32 TB

Table 1: Potential machines sizes based on the SSC micro architecture

A key advantage of a single architecture that scales from palm-top to super computers is program compatibility across the machines. Kernels, which are compiled for individual

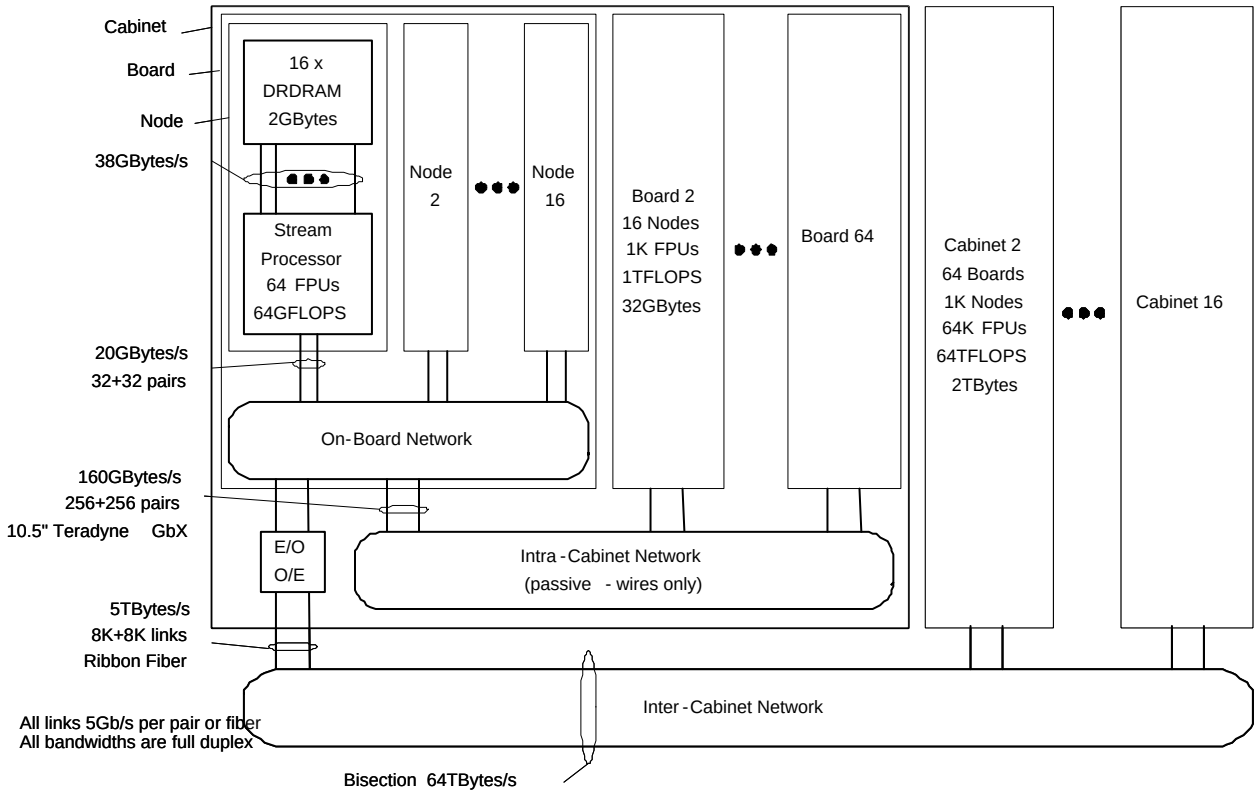

Figure 6: Organization of a streaming super computer

processors, should be compatible across all machine classes. However, applications will need repartitioning to efficiently scale to varying numbers of processors on the different classes of machines. This may be accomplished by a mix of compile-time and run-time techniques.

Figure 7 shows the micro architecture of a single stream processor (node) of the SSC. Note that the scalar processor (host) is integrated much more closely with the streaming resources compared to Imagine, which should help alleviate some of the “short stream effects” (i.e. overheads of communication between host and stream processor + the setup and tear down costs becoming dominant for short streams). The compute cluster and register file organizations are fairly similar to Imagine.

2.7 Open Issues

There are many open questions in the design of the SSC, some of which are listed here.

- Software issues:
  - Program transformation
  - Program mapping

Item	Cost	Per Node Cost
Processor chip	200	200
Router chip	200	50
Memory chip	20	320
Board/backplane	3000	188
Cabinet	50000	49
Power	1	50
Per-node total cost		976
$/GFLOPs (64/node)		15
$/M-GUPs (250/node)		4

Table 2: Cost estimates (parts cost only, no I/O)

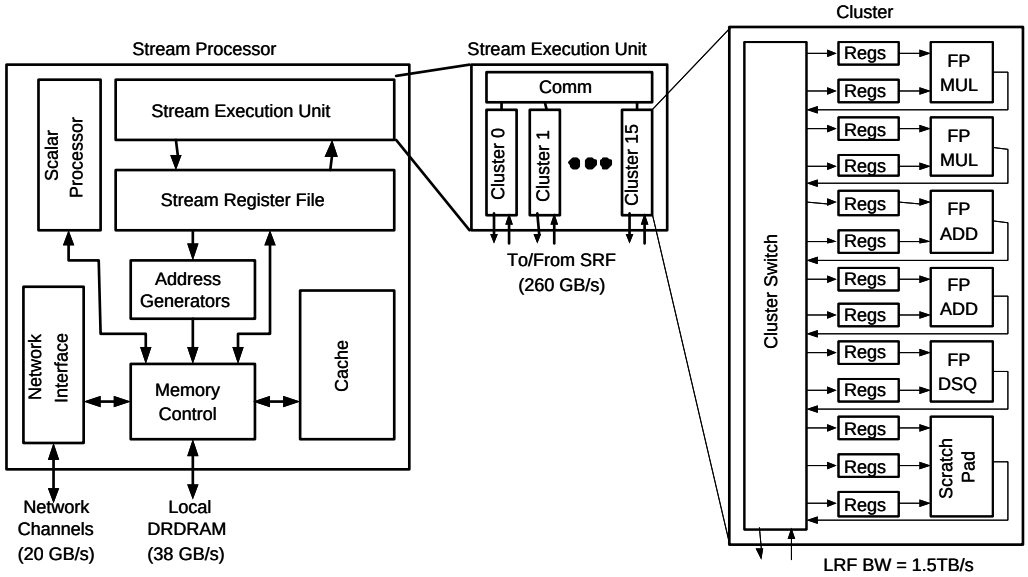

Figure 7: Streaming super computer node architecture

  - Bandwidth optimization
  - Conditionals
  - Irregular data structures

- Hardware issues:
  - Alternative stream models
  - Register organization
  - Bandwidth hierarchies

  - Memory organization
  - Short stream issues
  - ISA design
  - Cluster organization
  - Processor organization

3 Stream C and Kernel C Demonstration

Please refer to the demonstration code (not reproduced here) for more details.

3.1 Stream C Demo: Element-wise Multiply of Two Streams of Complex Numbers

Relevant Stream C commands:

- Declare and allocate streams
  - e.g.: `im_stream⟨cplx⟩ NAMED(in1) = newStreamData⟨cplx⟩ (length);`
    - `im_stream` keyword declares a stream
    - `NAMED()` directive makes the stream name available to the simulator
    - `cplx` is the data type of the stream (record type defined in .hpp file)
- Load stream contents from files to memory (note this load is to simulated DRAM, not to the SRF)
  - e.g.: `streamLoadFile(in1_fname.char_ptr(), "txt", "", in1);`
- Call kernel(s)
  - e.g.: `cplx_mul(in1, in2, out)`
    - `in1`, `in2`, and `out` are stream parameters
- Save output to a file
  - e.g.: `streamSaveFile(out_fname.char_ptr(), "txt", "E", out);`

3.2 Kernel C Demo 1: Element-wise Multiply of Two Streams of Complex Numbers

Observations:

- No loop-carried dependencies
- Schedule for single loop iteration inefficient (not enough ILP in one iteration to "pack" the schedule tightly).
- Providing loop unrolling and software pipelining hints to the scheduler results in a significantly better schedule (i.e. improves ILP by increasing the pool of instructions to be scheduled).

3.3 Kernel C Demo 2: Element-wise Multiply of Two Streams of Complex Numbers and Sum Them Up

Observations:

- Each cluster computes a partial sum of the elements that are assigned to its "local" SRF bank (done within loop)
- Partial sums from the 8 clusters are communicated via the inter-cluster communication network and are added using a tree sum (done outside the loop). See figure 8 for the tree sum communication pattern.

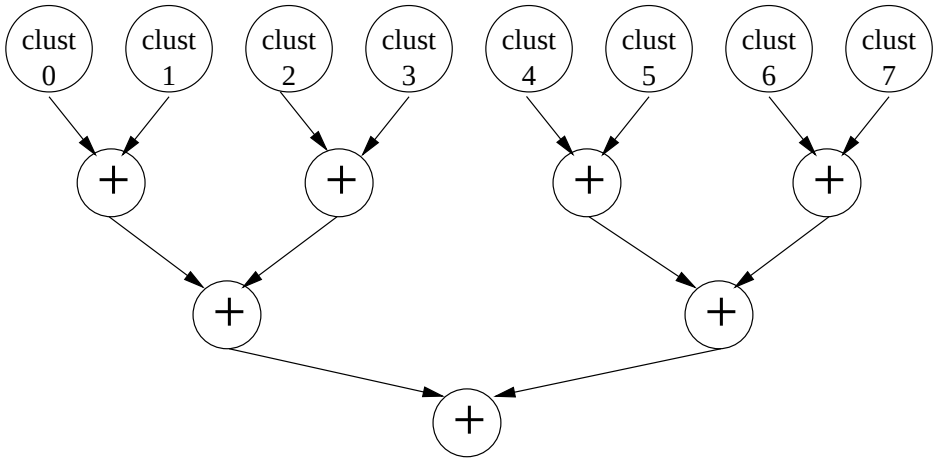

Figure 8: Tree sum

- The inter-cluster communication patterns are explicitly specified as 32-bit numbers, with each nibble specifying the data source for one of the clusters.

- The partial sum computation within the loop creates a loop-carried dependence, limiting the ability of the scheduler to optimize using automatic software pipelining and loop unrolling (even with hints provided).
- Manual loop unrolling in the Kernel C code leads to a much better schedule.
- A microcontroller variable (i.e. a scalar value that can be communicated between Imagine and host) is used to communicate the final sum back to the host.

4 Announcements

Review session at 3:00pm on Friday on 04/20/2002. The examples presented in class and the programming system will be discussed in more detail.