Raw/StreaMIT

Lecture #10: Tuesday, 7 May 2002
Lecturer: Bill Dally
Scribe: Sanjit Biswas and Dan Bentley
Reviewer: Mattan Erez

Announcements: Project proposals are due today, please send an e-mail with your group's status.

1 Language

1.1 Filter

The primary building block of StreaMIT is the filter. A filter has one input and one output. The syntax is Java-like. Each filter must have two functions, init() and work(). Init is called once, at the beginning. Work is called indefinitely and is the processing step of the system. Q. Does this imply the same number of in's and out's per function call? A. The ratio of data elements out to data elements in must be a compile time constant. Why can't this be dynamically determined? It has to do with the static routing of RAW, and will be discussed later.

1.2 Three Types of Filter

There are three types of filter: Pipeline, Split-Join and Feedback.
Pipeline:

Figure 1: Pipeline

Split-Join: In a split-join, the splitter can be one of round-robin(with weights), duplication and the null splitter (which does nothing). The joiner offers the choice of round-robin or null. Null is used for joining w/o splitting for sources or vice-versa for sinks.
Feedback: Feedback loops can be used for recursive functions. For instance, the Fibonacci function can be written as filter as peek(0) + peek(1). Note: This is useful for in-band feedback. For out-of-band or longer-range, use messaging. Messaging gives you semantics for delivering a message to a node at the proper time, in terms of element processed. The system probably does not handle speculative execution correctly, and you need to give the messaging system a realistic goal of when to deliver the message by.

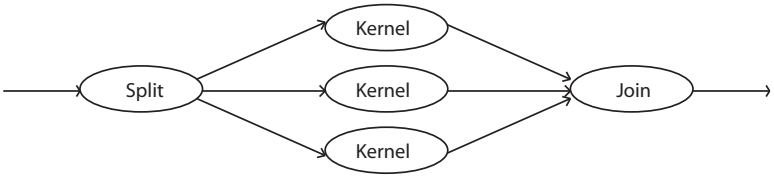

Figure 2: Split-Join

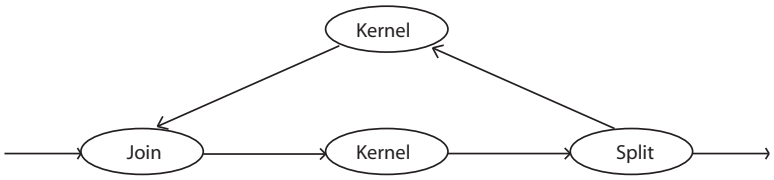

Figure 3: Feedback

2 Compilation

Their target was raw, but it is interesting to think how one might do this either on a single Imagine or an array of Imagines. Their example was an FM Radio. Taking that as ours... 1. Partition the filters: Deciding how many tiles per filter. This could result in fractional filters per tile. Note: In this paper, they have shown you some very nice transformations, but they are still only being done by hand. At some point, they hope to automate the process. 2. Iteratively solve: Find the bottleneck processor, then use a duplicator/split-join to load-balance. Continue until you run out of tiles. If some tiles are not busy, fuse them together. This leads to a nice formula for the speed-up.

$$MaximumSpeedup(w, c) = \frac{\sum_{i=1}^{N} w_i \cdot c_i}{MAX_i(w_i \cdot c_i)}$$

2.1 Amdahl's Law

At this point, there was a question raised about Amdahl's law and if this equation was somehow analogous. The answer was that this was not Amdah's law, and is a different view of execution time. Amdahl's law is seem in figure 4, whereas our view of execution is modeled in figure 5.

Thus, when the forks aren't the same length, a bottleneck exists. To change the execution time, we must shorten the length of the longest fork. In this case, we care about throughput, not latency (that's the beauty of streaming computation, it can hide latency).

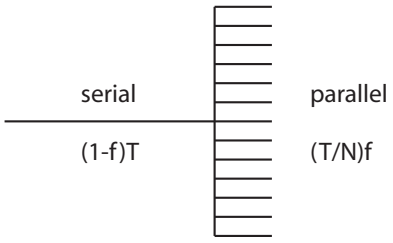

Figure 4: Amdahl execution

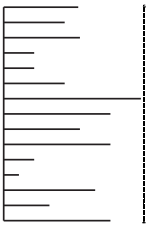

Figure 5: Parallel exec

3 Fusion Transformation

Two types of filter fusion techniques can be used to join two computationtally un-intensive nodes of the stream graph for improved processor utilization.

3.1 Vertical Fusion

Vertical fusion is fusing two in-series kernels.

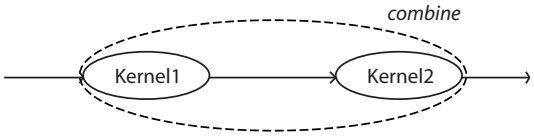

Figure 6: Vertical Fusion

When you would go to manually fuse two nodes, you would tend to merge the state yourself. For instance, if you have one node which duplicates and a second that sums up N things, you could write the code very simply. They don't. They don't even offer a system abstraction. Why not? The static scheduling facility of RAW makes this very difficult, because you must know how things affect scheduling at a different point. You only do vertical fusion when you know that neither of these nodes is going to be the bottleneck, at which point things have been laid out.

3.2 Horizontal Fusion

Horizontal fusion is fusing two parallel kernels.

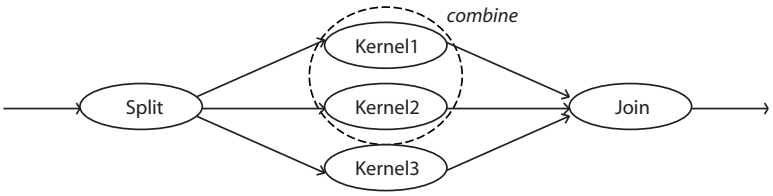

Figure 7: Horizontal Fusion

They perform this optimization. The paper only describes it for two filters, but it is easy to imagine this scaling up to an arbitrary number of kernels. In general, you want optimizations that either split bottlenecks or combine idle kernels to allow better utilization.

4 Fission Transformations

Two type of filter fission techniques can be used to split a computationally intensive node of the stream graph for improved load balancing. These transformations can be visualized as the reverse of the fusion transformations from the previous section.

4.1 Vertical Fission

Vertical fission involves splitting a single task into a pipelined set of multiple tasks. This is a potentially expensive transformations, as it involves carrying all necessary state forward (i.e. the live variables). This process has not yet been automated on StreamIt.

4.2 Horizonal Fission

Horizontal fission distributes a single filter across components of a SplitJoin. This transformation only works on "stateless" filters – basically those with no loop carried dependencies. Expressed in the context of StreamIt, these filters cannot contain fields written on one invocation of `work()` and read on a later invocation.

Examples: A filters which adds 3 to every element is stateless, but a summing filter is not. It should be noted that certain filters such as the Moving Average example on page 13 can be parallelized by duplicating overlapping portions of data. Since no data is carried across the sub, each node can compute an average for a specific range.

For filters which do not peek, we embed copied of the filter in a K-way RoundRobin SplitJoin, where the rates match the push/pop rates for the filter.

5 Reordering

5.1 Filter Hoisting

Allows the filter to be moved across joiner nodes, which is a useful transformation when load balancing. The hoisted filter must be stateless and the joiner's weights must be scaled to match.

5.2 Hierarchial

Intermediate levels of granularity may be obtained by breaking a SplitJoin into a hierarchial set then performing filter fusion. This can be used to obtain a finer degree of granularity for load-balancing purposes.

5.3 Synchronization Elimination

If a splitter is connected to a joiner with the same (or some same subsets) of weights, the SplitJoin can be removed and the components directly connected. This is useful when using libraries of components that use SplitJoins to process interleaved data streams.

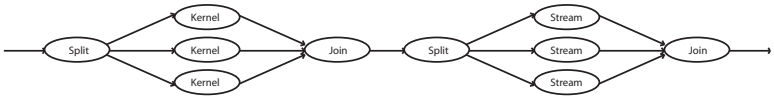

Figure 8: Synchronization Elimination (before)

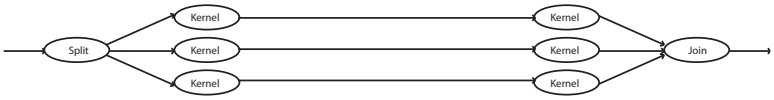

Figure 9: Synchronization Elimination (after)

6 Layout

Once all the partitioning transforms have been applied, and we have a load balanced graph that fits the number of tiles, nodes must be mapped to the computational tiles (i.e., put on Raw and communication network scheduled), with the goal of minimizing communication and synchronization.

The layout phase uses simulated annealing, which involves making a series of small changes followed by a probabilistic decision to find a global minimum. This requires the definition of architecture specific parameters, a cost function, a perturbation function and valid layouts. On Raw, this process is simplified, as the static network and communication

scheduler handle deadlock, all node assignments are legal and the perturbation function simply swaps random tiles.

Communication is the biggest concern, because while it costs nothing to go through an idle tile, there are large synchronization costs if the tile is busy or another communication is present. This is represented in the cost function by cubing the routing term:

$$cost(layout) = \sum_{(src,dst)\epsilon channels} \text{items}(src, dst)\cdot(\text{hops}(routingpath)+2\cdot\text{synch}(routingpath)^3)$$

Where $routingpath = route(layout(src), layout(dst))$

7 Communication Scheduling

We did not discuss communication scheduling in depth, but basically they avoid deadlock and starvation when scheduling the static network.

8 Results

Certain benchmarks (such as FFT) are deoptimized due to the very high cost of restructuring compared to the cost of the kernel itself.

Also, there appeared to be a discrepancy between MFLOPS and throughput. The initial ratings were obtained using an 8x8 structure, so the original filters could be mapped one-to-one. The optimized results were obtained by targetting a 4x4 processor and applying optimizations.

9 Questions and Thoughts

What is the benefit of space multiplexing over time multiplexing? More throughput and less latency since we don't see pipeline fill and flush effects. However, data locality is overlooked and accessing memory is difficult. Also, it's not clear which scales better.

9.1 Three Step Compilation

9.1.1 Partitioning

First handle the data structure by slicing up the texture map, etc. Then partition the streams (Raw provides static streams) and then partition the kernels (the primary focus of this paper).

9.1.2 Node Scheduling

Use software pipelining when scheduling the kernels, SRF and caches. The main idea is to hide latency by overlapping memory accesses with computation.

9.1.3 Kernel Scheduling

May result in more communication.