Stream Scheduling

Lecture #5: Thursday, 18 April 2002
Lecturer: Bill Dally
Scribe: Nate Hill

1 Tasks of a compiler

A compiler must map three resources in the hardware:

1. Computation - Arithmetic must be mapped to computation resources both in space and time
2. Storage - Hold data for temporal separations between computations
3. Communication - Move data between spatial separated computations

Mapping performed in compiler must be broken down into manageable chunks. For the stream compiler, this is done by allocation resources at both the program level (in units of stream-processing kernels) and the kernel level (in units of individual operations, such as addition, multiplication, inside each kernel). Note that conventional compilers can do the allocations/mappings at the kernel level, but the stream programming model allows the allocations/mappings to be done in the program/stream level as well.

A side note: the SRF is like a memory, though not exactly. The records in a stream in SRF can only be accessed sequentially, but multiple streams can be accessed at the same time.

2 Program level allocation

At the program level, part of the allocation that must be performed is allocating contiguous space in the stream register file for data. This allocation is for the life of the stream. The space is available again only after the kernel consuming it has finished entirely. Space for a stream cannot be partially reclaimed, because his would require us to know the rate at which records are being consumed, and the rate at which new records are being produced, and these two rates must be equal for this to work.

To handle arbitrarily large streams, the program must be broken down using strip mining or double buffering. Strip-mining is not done automatically. The compiler would analyze and suggest appropriate settings, but it must ultimately be done by hand. Also

note that, when strips are processed, there might be kernel states that have to be retained. This can be accomplished by either allocating persistent space in the scratchpad and/or dumping the states back to the SRF.

Also at the program level, there is partitioning to do to draw the kernel boundaries to be most efficient with the resources. This could also change the bandwidth requirements on the SRF and other resources. Currently this is optimized by the programmer writing the kernels.

To further look into this problem, think about this scenario: what if there are some small kernels that don't fully utilize all the functional units? Wouldn't it be neat the "fuse" these kernels together? This would reduce overhead and increase the working set. "Kernel fusion" may or may not increase demand on SRF: on one hand, it should increase demand on SRF because it now has to hold more input streams for more kernels; on the other hand, the intermediate results might never have to go out to SRF, so that can reduce demand. In any case, it's a open question how to do this and it may be a good project.

If this is so cool, why can't all kernels be simply fused into a single one? The strip size may be very small, but it should work! There are several problems. One is the iteration count problem: these kernels don't all have the same run time, so there's still a chance that the functional units are under-utilized as one kernel waits for another to finish. A bigger problem is conditionals (which is the topic for next paper).

3 Kernel level allocation

In the kernel level, one technique to increase parallelism is software pipelining. It is similar to strip mining but at the individual computation level. Within a single iteration of the loop, it does computations from different phases of the loop, allowing overlap between iterations of the loop.

The benefits of software pipelining include increasing the parallelism. This comes by expanding the current working set over several iteration of the loop. This larger working set must not exceed the storage available for allocation in order to be beneficial.

Similar results can be achieved by loop-unrolling, but software pipelining at least saves code spaces. Also, kernel executions and loading/saving streams from/to memory can be scheduled in an analogous manner, so that the long memory access times are hidden. Of course, doing this would increase demand on SRF, and as a consequence, strip size might have to be smaller. So, nothing's free.

4 Communication scheduling

Traditional architectures rely on a global, centralized register file for communication, in which everything (all registers' values) are available everywhere (available to any input of any ALU that need them). The problem with this is that the size/delay/power of the

register grows with N^3. An alternative is a clustered register file, in which outputs from execution units always go back to the same register file, and if another execution unit needs the result, an explicit copy must be performed. A third alternative is to make all states distributed, then schedule the communication between the distributed elements.

One advantage to distributed state and communication scheduling is that the registers have a fixed number of ports, allowing much better scalability. However, this adds complexity in that data has locality. Optimal scheduling requires understanding this locality and communication resources, which are shared.

One approach to communication scheduling is "list scheduling". Starting at time 0 the compiler allocates those operations that are ready. Then moving forward in time allocating each element as it becomes ready. The problem with this scheduling is that scheduling decisions can affect later communication availability. This can produce sub-optimal results or even make completing the scheduling impossible. This is because of a lack of lookahead to avoid "painting yourself into a corner".

The solution is to do the scheduling looking at the big picture. This allows the critical paths to be best scheduled and allows you to handle cases where you might otherwise get stuck.