Langages, Compilers, and Architectures for Stream Computing

Bill Mark

October 4, 2001

1 Summary

This document contains the follwing:

- A description of a HW architecture for a stream processor. I use the word ”architecture” to refer to the machine model presented to the low-level (assembly-language) programmer.
- A description of two possible strategies for a programming langage. They roughly correspond to my notion of the low-level and mid-level programming languages, but this is not completely fleshed out.
- A list of what makes compilation of a data-parallel language difficult
- A list of various transformations that a mid-level-language compiler might have to perform, and my assessment of whether they are easy, medium-difficult, or hard.

This document focuses primarily on a description of the underlying issues in language/compiler design, rather than on presenting a detailed language specification.

2 Hardware Assumptions

To provide a concrete basis for later discussions in this document, I will briefly state the hardware architecture that I am assuming. This hardware architecture is similar to the one that I suggested in an email in March, but with a few changes. Note that this hardware architecture need not correspond exactly to the programming model presented to the user.

- MIMD execution
- No general interprocessor communication (must use input sharing, communication via memory, or reduction)
- Inputs:
  - Inputs may be shared within some predefined neighborhood. (1D neighborhood definitely allowed; multi-dimensional neighborhoods probably allowed via strides)

  - Indirection is allowed on inputs (pointers to same stream or different stream)
  - No ”computed address” reads within a single kernel
  - One input stream is the ”primary” stream – this stream is used to invoke kernels. The primary stream can be a HW-generated stream of values 0..n; this serves as a means to ”create” a stream.

- Outputs:
  - Reduction operations (1D $\rightarrow$ scalar) are supported, for associative/commmutative operators. Ideally, reduction can occur at the tail of a kernel, but it could require a separate kernel execution.
  - Can output to multiple streams
  - Conditional outputs (1 input element $\rightarrow$ n output elements)
  - In SIMD mode, input stream order is preserved in outputs.
  - In MIMD mode, input stream order is *not* preserved in outputs. (This restriction could be removed, with additional HW cost)

- Scatters: Can scatter from a stream; possibly can scatter directly from a kernel output.

2.1 Possible additional capabilities

Most of these require some serialization.

- Ability to merge two streams on input, based on a key field.
- Incremental reduction operators (prefix reductions) on output.
- Ability to fall back to single-processor kernel execution, with stateful kernels and conditional inputs. If multiple kernels can be active at once in a pipelined manner, this serialization capability would be a general method of supporting reduction operators if the degree of parallelism was relatively small.
- Ability for a kernel to perform R/M/W access on a small region of memory (with address specified as input). Would require HW to detect conflicts and stall other elements if necessary.

2.2 Additional questions

- Does HW have ability to concurrently run more than one kernel, with a FIFO connecting them?
- Does HW have ability to perform dynamic balancing of processing resources when multiple kernels are running concurrently (if one of them is performing an expand).

3 Language

3.1 Two basic strategies

Fundamental capabilities and organization seem more important than precise syntax; I'll concentrate on the former.

I've thought about two major ways of organizing the programming environment:

- Three parts: Kernel code + flow graph specification + imperative code. The imperative code invokes the flow graphs in order. Imagine uses a variant of this model.
- Use a data-parallel language; everything gets extracted from this.

The first approach has a fairly clear correspondence to actual HW operations. Potentially, all kernels in a flow graph can be run concurrently; the flow graph does not allow gathers or scatters (except perhaps at the head and tail of the flow graph). However, the flow graph *can* include certain types of back edges. The imperative code runs on the non-parallel "host" machine, and might in fact just be a C program that invokes API routines for the other two levels. The primary disadvantage of this approach is that it somewhat artifically breaks the programming environment into three different pieces. The advantage of this approach is that it makes the compiler's job quite easy.

The second approach requires the compiler to perform some initial analysis to seperate code into the three categories discussed above. In some cases this analysis will be simple, but in other cases it may be difficult. In particular, it is unclear how to best represent back edges in a flow graph, such as iteration over a "not done" set of elements until it is empty. This type of operation can be expressed iteratively (as a while loop), or as a recursive function call. In either case, medium-difficult data-flow analysis is required to recognize such constructs.

An interesting question: Can we take the first approach, but with a slightly-more-unified programming interface? For example, a C program could include additional constructs processed by a metacompiler that define flow graphs, and indicate where to execute them.

I think that some variant of the first approach is appropriate for the low-level language; some variant of the second approach is appropriate for the mid-level language, *if* we decide that the compilation issues are tractable. We might initially settle for some hybrid of the two approaches for the mid-level language, to reduce the demands on the compiler.

There is another language design question that is relevant for either of the two approaches above: For multidimensional streams/arrays, should the language explicitly specify the traversal order? My inclination is that the low-level language should specify this traversal order (how??) while the mid-level language should not. The method of specifying traversal order must be general enough to allow the expression of traversal orders designed to improve cache blocking.

3.2 Details

Some capabilities that seem desirable:

- Ability to perform reads from arbitrary addresses; the system automatically splits the kernel at this point. (low-level language does not perform this split)
- Ability to read from bounded 1D, or multidimensional neighborhoods of current element.
- Ability to dereference incoming pointers, to the same stream or to a different stream. Chained dereferences as interpreted as a read from an arbitrary address, and thus cause the system to automatically split the kernel. (low-level language does not perform this split)
- For low-level language, ability to specify traversal order of multi-dimensional arrays/streams in a fairly general manner.

4 Compiler

The metacompilation framework appears to be very powerful for performing rule-based transformations, but is not designed to perform NP-class optimizations. It is likely that major effort will have to be expended to perform NP-class optimizations. This effort could be directed towards dramatically enhancing the metacompiler, or towards building a separate tool to perform these optimizations.

4.1 What makes compilation difficult?

When thinking about the system design, it's easy to focus on "simple" cases, and overlook a number of language and application features that make compilation of a data-parallel mid-level language difficult. The following is a list of some of the key causes of difficulties.

- Multidimensional arrays:
  - How to lay them out in memory?
  - How to traverse them (execution order; blocking issues; etc.)?
  - Analysis of dependencies between cascaded kernels; this is made more difficult because of the memory layout and traversal issues. It's especially hard if kernels depend on 2D input neighborhoods, rather than single elements.
  - How do multidimensional arrays interact with kernel granuarity? E.g. in 2D traversal should each dimension be performed by a separate kernel? Consider the implementation of convolution on Imagine.
- Granularity:

  - Data-parallel operations will often be nested. At what granularity do we package operations into a kernel? Simple example: a data-parallel operation on a four-vector probably shouldn't be treated as a stream operation.
  - For MIMD kernels, what control constructs are placed within a kernel, and which are placed outside a kernel?

- Hierarchies of parallelism – related to granularity
  - How to execute multiple levels of parallelism at once (e.g. stream + task; e.g. cascaded kernels). This problem is more important on a streaming supercomputer than it is on a single-chip stream machine.

- Iterative and recursive computations:
  - How to identify constructs that map to back edges in the data flow graph?

- Dynamically-sized objects – one or more varying dimensions:
  - Memory allocation
  - Cache blocking
  - On hardware that can only run one kernel at a time, how do we schedule cascaded kernels to "emulate" a FIFO between them?
  - On hardware that can run more than one kernel at a time, how do we allocate processors to do load balancing? This might need to be done at run time if the first kernel is doing an "expand".
  - etc.

- Irreglar data structures – or more generally, pointers/indices
  - Difficult to analyze for dependencies
  - May require HW-managed caches (tagged caches), to benefit from temporal locality in memory accesses.

- Different types of collections
  - Three types: Ordered vs. unordered vs. array
  - Interaction with MIMD kernels – should these kernels destroy ordering?
  - How to expose all of this in language?

- Scattering/permute – how to handle ordering in many $\rightarrow$ one case?

- Algorithms that are intrinsically global – e.g. Sorting

4.2 Easy vs. hard compiler transformations

4.2.1 Easy transformations

- Split/merge kernels to handle computed addresses for memory reads
- Identifying sequences of operations that can be combined into a single HW kernel. (note that this problem becomes somewhat more difficult once MIMD kernels are allowed... can't just treat any basic block as a kernel).

4.2.2 Medium transformations/operations

- Efficiently splitting kernels that exceed resource limits (e.g. registers, instructions, max input streams, max output streams, etc.)
- Blocking of multi-dimensional array accesses to maximize cache utilization
- Efficient support for variable length streams (or more generally, arrays with one or more dimensions that vary in size). e.g. blocking computations to fit intermediate streams into on-chip stream buffer. Note that running multiple kernels concurrently could help this problem; that's what graphics HW does.
- Identifying sequences of operations that can be partially pipelined (by implementing them as separate kernels that have only partially-overlapping memory accesses)
- Identifying "while" loops and recursive function calls that can be mapped to back edges in kernel-data flow graph.
- Distinguishing between conditional constructs that belong within a kernel, and those that must operate at either the flow-graph or imperative level.
- Identifying the correct granularity of operations to place in a kernel vs. a series of kernels vs. a task vs. . . . For example, a data-parallel operation on a 4-element array should probably be performed *within* a kernel, if this operation is nested within larger-granularity data-parallel operations.

4.2.3 Really-hard transformations

- Rewrite algorithms – e.g. make choice between different algorithms to solve same problem.
- Converting 100% functional code into efficient imperative code. [Reason: User has no method to provide guidance to compiler]

5 General questions

- Do we support dynamically-allocated streams?
- Do we support variable-sized streams? (possibly only one dimension is variable?)
- ...