Mapping Vector Codes to Stream Processor (Imagine)

Mehdi Baradaran Tahoori
Paul Wang Lee
June 4th, 2002
Outline

- Motivation
- Problem Statement
- Simulation Results
- Conclusion
Motivation

- Large volume of vector code in existence
  - Arithmetic intensive

- Much existing research on vectorization
  - Vectorizing compilers, etc.

- Stream programming
  - Intermediate data
    - Producer-consumer locality
    - Shorter lifetime than in vector processor
Problem Statement

Efficient mapping of vector codes to stream processor

- Pseudo vector code
  - Not focusing on syntax
- Focus on specific hardware
  - Imagine architecture
  - Imagine programming model
  - Stream C & kernel C
- No performance evaluation in Brook
Goals

● Maximize resource utilization

● Minimize memory bandwidth requirements
  ◆ SRF ↔ LRF
  ◆ SRF ↔ μC

● Minimize inter-cluster communications
  ◆ Specially for vector reduction operations
    ◆ Inner-product, matrix × vector, ...
Approach

- Implementation in KernelC & StreamC
  - Cycle accurate simulation
    - Various representative code snippets
    - Various record sizes
    - Various kernel granularity
  - Considering realistic settings
- Observations through simulation
  - Interpret results
  - Look for rules
    - Can be applied to mapping strategy
Partitioning

- Modulo Data
  - Stream element size
    
    record vect \{float v0, ..., vn;\}
    kernel VADD(istream<vect> A, istream<vect> B, ostream<vect> C)

- Modulo Operation
  - Kernel granularity
    \[ E[15:0] = C[15:0] \cdot D[15:0] \]
    
    kernel VADD(A,B,C)
    kernel VMUL(C,D,E) kernel VADD_MUL(A,B,D,E)
Effect of Record Size on Scheduling

- Better scheduling with larger record sizes
- Unrolling has the same effect of increasing record size

Scheduling (normalized to one element)
Total Execution Time

- Not as expected!!!
Why Worse?
Reason

Detailed cycle count

- Kernel
- SRF->uc
- Ucode
μCode

- μCode is first loaded to SRF
- Then loaded from SRF to μController
- Record size $\uparrow \Rightarrow \mu$Code size $\uparrow$
- μCode cost can be *amortized*
  - reusing the same kernel

- Less of an issue for larger data sets
Amortized $\mu$Code

10 ADD kernels

- Execution cycles vs. record size
  - 1 record size: 60000 cycles
  - 8 record size: 60000 cycles
Kernel Granularity

- **Extreme cases:**
  - Each operation in a separate kernel
  - All operations in one big kernel

![Mathematical expressions and diagrams](image-url)
Serial Computations

- 256 data set
- No software pipelining in kernel scheduling

$V_o = \frac{(V_1 + V_2)}{V_3} \cdot V_4$
Non-serial Computations

- 256 data set
- No software pipelining in kernel scheduling

\[ V_0 = (V_1 + V_2) \cdot (V_3 / V_4) \]

![Graph showing the relationship between record size and execution cycles for 1 kernel and 3 kernels.](image)
More Non-serial Computations

- 256 data set
- No software pipelining in kernel scheduling

$$V_o = (V_1 + V_2) \cdot (V_3 + V_4)$$
Two cases

- For serial computation
  - Smaller kernels with smaller record sizes
    - Best performance

- For non-serial (parallel) computation
  - Bigger kernels always better
    - Better resource utilization
More Simulations

- Computational intensive operations
  - Heavy loops
    - Carry independent
  - Large Matrix by Vector manipulation
- Effect of software pipelining
  - Better resource utilization
Non-Serial Computation

Loops of $(V1 + V2) \cdot (V3 + V4)$
Serial Computation

Loops of \((V1 + V2) / V3 \cdot V4\)

![Graph showing execution cycles for different record sizes and kernel configurations.](image-url)
Larger Case

- Two Dataflows
  - Balanced tree
  - Fully dependent
Kernel Fusion

- Essentially kernel fusion
  - Merge 1-op kernels
Parallel Chain

![Graph showing normalized runtime versus max #Ops per Kernel for different key sizes (256, 2048, 4096, 8192). The x-axis represents the max #Ops per Kernel, ranging from 1 to 5, and the y-axis represents the normalized runtime, ranging from 0 to 20. The graph shows a decreasing trend in normalized runtime as the number of operations increases.]
Kernel size

- Larger kernels are better for reasonable data size
  - More ops to schedule
  - Once there are enough ops, no more benefit
  - But, for data size comparable to SRF, large kernels still better

- Limits to kernel size
  - LRF size limit
  - Limit to number of streams per kernel
Conclusion

- Explored basic issues of mapping vector code to stream code
  - Mostly confirmed intuition
  - Found a few issues we did not consider

- Next logical step: set of criteria for kernel fusion
  - Need to satisfy many constraints
  - Best solution may be impractical to find
  - Set of heuristics would probably suffice