Stream Architectures – Programmability and Efficiency

Mattan Erez
Computer Systems Laboratory

SoC 2004 – November 17, 2004
The Performance Issue

Wireless communications (3G, UWB, …)
- Higher data rates
- More complex air interfaces

Video processing (DTV, security, …)
- Compression
- Anomaly detection
- Feature extraction

Image processing (cameras, copiers, …)

Wire-line (VoIP, DSL, cable)

Embedded signal and image processing tasks are becoming more demanding requiring 100s to 1000s of GOPS
The Efficiency Issue

- Battery life and heat – 100s of MOPS/mW (GOPS/W)
- Commodity and volume – 10s of GOPS/$

Energy consumption, power dissipation, and cost are critical
The Programmability Issue

- Multiple modes
- Evolving standards
- Evolving features, differentiation
- Design/tooling costs

Programmability reduces cost, enables differentiation, and adaptation, and improves time-to-market
Stream architectures achieve ASIC-like performance and efficiency with CPU style programmability.
Outline

• DSPs, ASICs, and FPGAs
• Stream Architectures
  – Efficient hardware
  – Capable compiler
• Imagine
• Stream scalability from blocks to supercomputer
  – SPI SP-1
  – Merrimac
• Conclusion
ASIC: high performance and low energy but expensive  
DSP: programmable but low performance and high energy  
FPGA: high performance but not efficient

<table>
<thead>
<tr>
<th>Feature</th>
<th>ASIC</th>
<th>DSP</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOPS/W</td>
<td>50 – 200</td>
<td>&lt;10</td>
<td>2 – 10</td>
</tr>
<tr>
<td>GOPS/$</td>
<td>2 – 10</td>
<td>~0.1</td>
<td>~1</td>
</tr>
<tr>
<td>Peak GOPS</td>
<td>Up to 1000</td>
<td>&lt;10</td>
<td>Up to 500</td>
</tr>
<tr>
<td>Design Cost</td>
<td>$15M</td>
<td>$1M programming</td>
<td>$5M all but masks</td>
</tr>
<tr>
<td>Programmability</td>
<td>Fixed</td>
<td>programmable</td>
<td>reconfigurable</td>
</tr>
</tbody>
</table>

ASIC, DSP, and FPGA
## Bandwidth Dominates Energy Consumption

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy (0.13µm)</th>
<th>Energy (0.05µm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b ALU Operation</td>
<td>5pJ</td>
<td>0.3pJ</td>
</tr>
<tr>
<td>32b Register Read</td>
<td>10pJ</td>
<td>0.6pJ</td>
</tr>
<tr>
<td>Read 32b from 8KB RAM</td>
<td>50pJ</td>
<td>3pJ</td>
</tr>
<tr>
<td><strong>Transfer 32b across chip (10mm)</strong></td>
<td><strong>100pJ</strong></td>
<td><strong>17pJ</strong></td>
</tr>
<tr>
<td>Execute a uP instruction (SB-1)</td>
<td>1.1nJ</td>
<td>130pJ</td>
</tr>
<tr>
<td><strong>Transfer 32b off chip (2.5G CML)</strong></td>
<td><strong>1.3nJ</strong></td>
<td><strong>400pJ</strong></td>
</tr>
<tr>
<td>Transfer 32b off chip (200M HSTL)</td>
<td>1.9nJ</td>
<td>1.9nJ</td>
</tr>
</tbody>
</table>

1:20:260 local to global to off-chip ratio today
1:56:1300 in 2010

Locality is key to achieving energy efficiency
To Exploit VLSI Technology We Need:

- **Parallelism**
  - 10s of FPUs per chip (hundreds/board thousands/system)

- **Latency tolerance**
  - To cover hundreds of cycles remote memory accesses

- **Locality**
  - To match 20Tb/s ALU bandwidth to ~200Gb/s off-chip bandwidth

Arithmetic is cheap, global bandwidth is expensive
Local << global on-chip << off-chip << global system
Why do Special-Purpose Processors Perform Well?

Lots (100s) of ALUs (parallelism)被馈入由专用的电线/记忆体（locality）

ASICs rely on parallelism and locality
Programmable Processors Ignore Locality

Data and instructions communicate through global structures
Stream processors exploit locality and parallelism
Kernels exploit instruction-level parallelism (ILP) and expose short-term producer-consumer locality.

Kernels can be partitioned across chips to exploit task parallelism.

Streams expose long-term producer-consumer locality and data-level parallelism.

Image 0

convolve

Image 1

convolve

Gaussian for noise reduction

Laplacian for sharpening

SAD

Depth Map

Stream model exploits parallelism and locality without complexity of traditional parallel programming.
Stream Architecture Makes Communication Explicit – Exploits Parallelism and Locality

Register hierarchy and data-parallel execution enable high performance and efficiency.
Stream Architecture Exploits ILP and DLP for Efficient Instruction Supply

VLIW within cluster and SIMD across clusters
Statically scheduled switch and register files within a cluster
Producer-Consumer Locality in the Depth Extractor

<table>
<thead>
<tr>
<th>Memory/Global Data</th>
<th>SRF/Streams</th>
<th>Clusters/Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>row of pixels</td>
<td>Convolution (Gaussian)</td>
</tr>
<tr>
<td></td>
<td>previous partial sums</td>
<td>Convolution (Laplacian)</td>
</tr>
<tr>
<td></td>
<td>new partial sums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blurred row</td>
<td></td>
</tr>
<tr>
<td></td>
<td>previous partial sums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new partial sums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sharpened row</td>
<td></td>
</tr>
<tr>
<td></td>
<td>filtered row segment</td>
<td>SAD</td>
</tr>
<tr>
<td></td>
<td>filtered row segment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>previous partial sums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new partial sums</td>
<td></td>
</tr>
<tr>
<td></td>
<td>depth map row segment</td>
<td></td>
</tr>
</tbody>
</table>

1:23:317 MEM:SRF:LRF word access ratio
Latency Tolerance through Pipeline Parallelism

- Stream memory operations amortize memory latency
- SRF is staging area for memory I/O and decouples memory accesses from computation
  - Strip-mining partitions the computation
  - Software pipelining overlaps kernel execution with I/O to store previous results and load data for next strip

Memory access times can be hidden by useful work
Programming tools map “C” code to stream processor – optimizing communication locally and globally.

Rely on stream model to expose locality and parallelism. Utilize decoupling of memory and execution for effective optimization.
Coded in “C”, Familiar Development Environment

Step 1
StreamC/KernelC reflect hardware strengths

Step 2

Step 3

Step 4
Communication Scheduling Achieves Near Optimum Kernel Performance

7x7 convolution kernel from depth extraction application

(Above) Single iteration schedule
(Right) Software pipelining shown

Staging memory through SRF leads to predictable latencies and allows effective VLIW communication scheduling
Stream Scheduling Example

- ALU Clusters (left-most column) fully occupied
- Operations from current batch (yellow) overlapped with operations from other iterations (blue)

Software pipelining can hide memory latencies
Stream Scheduling Reduces Bandwidth Demand by up to 12:1 Compared to Caching

- **Stream program**
- **SRF allocation**

Software managed SRF allows compiler to maximize locality
The Imagine prototype demonstrates streams on signal and image processing applications
HW: Stream Processor Architecture (8 cluster example)

- Streams stored interleaved in Stream Register File (SRF) Blocks
- Kernels executed on SIMD/VLIW arithmetic clusters

48 FP ALUs  25.4 GOPS  5.79W  16x16mm 0.18μm process
Imagine Bandwidth Hierarchy Exploits Locality

>90% of data movement is from local registers
Prototype HW and SW

- Prototype of Imagine architecture
  - Proof-of-concept 2.56cm² die in 0.18um TI process, 21M transistors
  - Collaboration with TI ASIC

- Dual-Imagine development board
  - Platform for rapid application development
  - Test & debug building blocks of a 64-node system
  - Collaboration with ISI-East

- Software tools based on Stream-C/Kernel-C
  - Stream scheduler
  - Communication scheduling

- Many Applications
  - 3 Graphics pipelines
  - Image-processing apps – depth, MPEG
  - 3G Cellphone (Rice)
  - STAP
  - Network processing – IPv6 and VPN (NC State)

Imagine demonstrates effectiveness of stream processing
Stream architectures are scalable from IP blocks to supercomputers.

SPI commercializes streams for embedded applications. Merrimac streaming supercomputer increases range of stream processing.
A Commercial Stream Processor – SP1: 1.3 TOPS, 12 Watts, Q105 Tapeout

- 64 clusters x 10 16-bit MulAdds
  - 1GHz at 1.0V, 12W
  - 670MHz at 0.8V, 5W
- Efficiency of ASIC
  - 640 billion multiply-adds per second (1.28 TOPS)
  - 6 pJ per operation (0.8V)
    - 160MOPS/mW
    - 116 mm²
- Tapeout in Q1 ’05
  - 8 cluster version at 0.13µm
- Modular and Scalable
  - 8 clusters, 20mm², 160GOPs, 1.5W
  - 150mW, 24GOPs at 150MHz 0.8V

Tuned for signal/image processing – not a research platform
Power Dissipation

- Imagine (0.18 μm – 48 FP ALUs)
  - 3.1 W, 132 MHz, 1.5 V (meas.)
- Power dissipation is dominated (>90%) by very predictable sources
  - RFs
  - ALUs
  - switches between ALUs
  - clocks

Stream processing is efficient: ~40% of energy in ALUs
Stream processing: OPS/W competitive with ASIC – Programmability of DSP

MOPS/mW

1000

100

10

1

ASICS

Array Processors

FPGAs

DSPs

STREAM PROCESSORS

Specialized

Programmable in “Verilog”

Programmable in “C” SW Upgrades

Improved Flexibility & Productivity

Stream and ASIC rely on parallelism and locality
Stream processors: more OPS/mm² than an ASIC

Programmability allows for semi-custom one-time design
A streaming supercomputer exploits
the arithmetic density of VLSI to realize
an efficiency of $6$/GFLOPS

Capability AND Capacity
A PFLOPS machine with only 8,192 nodes (or less)
A TFLOPS workstation for < $20,000 (parts cost)
Architecture of a Streaming Supercomputer

Scalable from 2-TFLOP workstation to 2-PFLOP supercomputer
Merrimac Processor is Capable and Power and Area Efficient

- **64 64-bit MULADD FPUs**
  - Arranged in 16 clusters
- **Capable memory system**
- **Designed for reliability**
- **1 GHz in 90nm**
  - 128 GFLOPS
- **Area efficient**
  - ~150mm² in 90nm
  - Pentium 4 is ~120mm² in 90nm but only 6.4 GFLOPS
- **Efficient at ~25W**
  - Pentium 4 is 100W
  - 28% of energy in ALUs

Merrimac processor is tuned for scientific computing
Merrimac Compute Cluster
Addresses Application Needs

- 4 64-bit MULADDs
- Large LRFs
  - 96 words per FPU
- Accelerated iterative ops (throughput of 1 op/cycle)
  - 1/x
  - 1/sqrt(x)
- Indexed SRF
  - Capture more locality
  - Stencils
- Conditional streams
  - Variable rate streams

Merrimac processor is tuned for scientific computing
Merrimac Features a High Bandwidth Memory System with Latency Tolerance

- High bandwidth using commodity DRAM
  - 8 XDR-DRAM interfaces for 64 GB/s peak
  - Memory address scheduling for increased utilization

- Stream cache amplifies bandwidth
  - Not crucial for performance
  - Low hit rate still beneficial
  - Can cache remote data

- Stream memory operations
  - Amortize memory latency
  - Amortize overheads

High bandwidth with commodity parts and amplification
Merrimac is Tuned for Large-Scale Scientific Computing

- Complex record-based addressing modes
  - Gather
  - Scatter
  - Strided

- Scatter-add is a data parallel atomic fetch&op
  - allows efficient superposition and histogram

- Single global address space
  - Efficient single word access through high-radix interconnect
  - Ease of programming

Gather/scatter and atomic scatter-add for superposition
Single global address space across thousands of nodes
Scientific Streamed Applications

- **StreamFLO** is a streaming version of FLO82, [Jameson], for the solution of the inviscid flow around an airfoil
  - Uses a cell centered finite volume formulation with a **multi-grid** acceleration to solve the 2D Euler equations
- **StreamFEM** implementation of the Discontinuous Galerkin (DG) Finite Element Method (FEM) (Tim Barth, NASA)
  - 2D or 3D triangulated domains
  - Increasingly complex PDEs
    - Scalar advection (1 PDE), Euler (4 PDEs), Magnetohydrodynamics (6 PDEs)
  - Piecewise polynomial function
    - Constant (1 dof), Linear (3 dof), Quadratic (6 dof), Cubic (10 dof)
- **StreamMD** molecular dynamics simulation
  - Box of water molecules
  - Electrostatic and Van der Waals interactions
  - Gridded to accelerate approximate force calculation

Many scientific applications have parallelism and locality
## Summary of Application Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Sustained GFLOPS</th>
<th>FP Ops / Mem Ref</th>
<th>LRF Refs</th>
<th>SRF Refs</th>
<th>Mem Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamFEM3D (Euler, quadratic)</td>
<td>31.6</td>
<td>17.1</td>
<td>153.0M (95.0%)</td>
<td>6.3M (3.9%)</td>
<td>1.8M (1.1%)</td>
</tr>
<tr>
<td>StreamFEM3D (MHD, constant)</td>
<td>39.2</td>
<td>13.8</td>
<td>186.5M (99.4%)</td>
<td>7.7M (0.4%)</td>
<td>2.8M (0.2%)</td>
</tr>
<tr>
<td>StreamMD (grid algorithm)</td>
<td>14.2*</td>
<td>12.1*</td>
<td>90.2M (97.5%)</td>
<td>1.6M (1.7%)</td>
<td>0.7M (0.8%)</td>
</tr>
<tr>
<td>GROMACS</td>
<td>38.8*</td>
<td>9.7*</td>
<td>108M (95.0%)</td>
<td>4.2M (2.9%)</td>
<td>1.5M (1.3%)</td>
</tr>
<tr>
<td>StreamFLO</td>
<td>12.9*</td>
<td>7.4*</td>
<td>234.3M (95.7%)</td>
<td>7.2M (2.9%)</td>
<td>3.4M (1.4%)</td>
</tr>
</tbody>
</table>

* The low numbers are a result of many divide and square-root operations

Applications achieve high performance and make good use of the bandwidth hierarchy.
Conclusion
Stream Processors vs. ASICs

Stream Processors
- Optimized design of cluster
  - \(>500\) GOPS/mm\(^2\)
  - \(~150\) GOPS/W
- Same hardware supports multiple modes of operation
  - Text vs. image
  - Multiple codecs
  - Multiple air interfaces
  - Translates to GOPS/mm\(^2\)
- Software-only development for new application
  - Faster time-to-market
  - Add features
  - Track standards

ASICs
- Exactly match required bit-width and mix of ALUs
  - \(1000\) GOPS/mm\(^2\), \(<200\) GOPS/W
- No instruction overhead
  - Very low with streams
- Can provide ‘unusual’ operations
  - E.g., Galois field ops
- But expensive
  - Design
  - Tooling
  - Verification

Efficient, flexible, programmable, and cost-effective
Summary

• Stream processors provide the performance and efficiency of an ASIC, but are programmable in “C”
  – A stream program exposes parallelism and locality
  – The register hierarchy makes communication (data motion) explicit
  – A stream compiler maps the program to the hierarchy to exploit locality

• Working silicon today
  – Imagine prototype demonstrates high efficiency on many applications
  – SP1 will provide up to 1.3TOPS, 12W, 160MOPS/mW

• Working software tools today
  – StreamC/KernelC efficiently mapped to hardware
  – No assembly, automatic staging of data

• Enables programmability in demanding applications
  – Better algorithms, evolving standards
  – Multiple operating modes, differentiate products

Why would anyone use an ASIC/ASSP?