Stream Processing for High-Performance Embedded Systems

William J. Dally
Computer Systems Laboratory
Stanford University

HPEC
September 25, 2002
Outline

• Embedded computing demands high arithmetic rates with low power
• VLSI technology can deliver this capability – but microprocessors cannot
• Stream processors realize the performance/power potential of VLSI while retaining flexibility
Embedded systems demand high arithmetic rates with low power.

For $N=10$, $BW=100\text{MHz}$, $S=16$, $B=4$, about $500\text{GOPs}$
VLSI provides high arithmetic rates with low power – microprocessors do not

PowerPC G4
95mm$^2$ $\sim$1nJ/op

32b adder + RF, 512 x 163 tracks
205$\mu$m x 65$\mu$m $\sim$ 0.013mm$^2$
$\sim$5pJ/op

Area 7300:1, Energy 200:1, Ops 4:1
VLSI provides high arithmetic rates with low power – microprocessors do not

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

300: 20: 1 off-chip to global to local ratio in 2002
1300: 56: 1 in 2010
Why do Special-Purpose Processors Perform Well?

Lots (100s) of ALUs

Fed by dedicated wires/memories
Care and Feeding of ALUs

‘Feeding’ Structure Dwarfs ALU
Stream Programs Expose Locality and Concurrency

Kernels exploit both instruction (ILP) and data (SIMD) level parallelism. Kernels can be partitioned across chips to exploit task parallelism.

Streams expose producer-consumer locality. The stream model exploits parallelism without the complexity of traditional parallel programming.
A Bandwidth Hierarchy exploits locality and concurrency

- VLIW clusters with shared control
- 41.2 32-bit floating-point operations per word of memory BW
## 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><img src="image1.png" alt="Image of memory/global data" /></td>
<td><img src="image2.png" alt="Image of SRF/Streams" /></td>
<td><img src="image3.png" alt="Image of clusters/kernels" /></td>
</tr>
</tbody>
</table>

- **Row of pixels**
- **Previous partial sums**
- **New partial sums**
- **Blurred row**
- **Previous partial sums**
- **New partial sums**
- **Sharpened row**
- **Filtered row segment**
- **Filtered row segment**
- **Previous partial sums**
- **New partial sums**
- **Depth map row segment**

**Convolutions**:
- Convolution (Gaussian)
- Convolution (Laplacian)

**SAD**: $1 : 23 : 317$
A Bandwidth Hierarchy exploits kernel and producer-consumer locality

<table>
<thead>
<tr>
<th></th>
<th>Memory BW</th>
<th>Global RF BW</th>
<th>Local RF BW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Depth Extractor</strong></td>
<td>0.80 GB/s</td>
<td>18.45 GB/s</td>
<td>210.85 GB/s</td>
</tr>
<tr>
<td><strong>MPEG Encoder</strong></td>
<td>0.47 GB/s</td>
<td>2.46 GB/s</td>
<td>121.05 GB/s</td>
</tr>
<tr>
<td><strong>Polygon Rendering</strong></td>
<td>0.78 GB/s</td>
<td>4.06 GB/s</td>
<td>102.46 GB/s</td>
</tr>
<tr>
<td><strong>QR Decomposition</strong></td>
<td>0.46 GB/s</td>
<td>3.67 GB/s</td>
<td>234.57 GB/s</td>
</tr>
</tbody>
</table>
Bandwidth Demand of Applications
Local registers increase effective size and bandwidth of SRF

• ~90% of live variables are captured in local registers
• Only 10% of live variables need be stored in stream register file
• Fixed-size SRF is effectively 10x the size of a VRF that must hold all live variables
• Bandwidth into FPUs is 10x the SRF bandwidth
Cluster Occupancy > 80%
Performance demonstrated on signal and image processing

![Graph showing performance of various applications and kernels.](image)
Prototype

• Prototype of Imagine architecture
  – Proof-of-concept 2.56cm$^2$ die in 0.18um TI process, 21M transistors
  – Collaboration with TI ASIC
  – Runs all benchmarks at 240MHz

• Dual-Imagine development board
  – Platform for rapid application development
  – Test & debug building blocks of a 64-node system
  – Collaboration with ISI-East
Imagine is programmed in “C” at two levels

- **Streams:**
  - Sequences of records

- **Kernels:**
  - Functions that operate on streams
  - Written in KernelC
  - Compiled by kernel scheduler

- **Stream program:**
  - Defines streams, control- and data-flow between kernels
  - Written in StreamC and C++
  - Compiled by stream compiler
Simple example

• **StreamC:**

```c
void main() {
    Stream<int> a(256);
    Stream<int> b(256);
    Stream<int> c(256);
    Stream<int> d(1024);
    ...
    example1(a, b, c);
    example2(c, d);
    ...
}
```

• **KernelC:**

```c
KERNEL example1(  
    istream<int> a,  
    istream<int> b,  
    ostream<int> c)  
{
    loop_stream(a) {
        int ai, bi, ci;
        a >> ai;
        b >> bi;
        ci = ai * 2 + bi * 3;
        c << ci;
    }
}
```
Communication scheduling achieves near optimum kernel performance

7x7 convolution kernel from depth extraction application

(Above) Single iteration schedule

(Right) Software pipelining shown
Stream scheduling reduces bandwidth demand by up to 12:1 compared to caching

Stream program

- Matrix
- Viewport
- Lights
- Shader
- Span Prop
- Span Convert
- Rasterize
- Render
- Compact
- Z Compare
- Depth buffer
- Hash
- Sort
- Merge
- Viewport

Open GL graphics pipeline

Current DSP programmers attempt to stage data in this manner by hand

SRF allocation
We have developed...

- A *stream architecture* that exploits locality and concurrency
  - Keeps 99% of the data accesses on chip
  - Aligned accesses to SRF
  - Enables efficient use of large numbers (100s) of ALUs

- Imagine: a prototype *stream processor* that demonstrates the efficiency of stream architecture
  - Working in the lab at 240MHz
  - 9.6GFLOPS, 19.2GOPS, 6W
  - Programmed in “C”
  - Sustains ~5GOPS/W at 1.2V (200pJ/OP)

- and demonstrated image-processing, signal processing, and graphics applications on the Imagine stream processor
Stream processing can be applied to scientific computing

- Extensions to architecture
  - 64b floating point – 100GFLOPS/chip
  - Support 2-D, 3-D, and irregular data structures
    - Stream cache
    - Indexable SRF

- Estimates suggest we can achieve
  - < $20/GFLOPS
  - < $10/M-GUPS
Conclusion

- Streams expose locality and concurrency
  - Concurrency across stream elements
  - Producer/consumer locality
  - Enables compiler optimization at a larger scale than scalar processing
- A stream architecture exploits this to achieve high arithmetic intensity (arithmetic rate/BW)
  - Keeps most (>90%) of data operations local (544GB/s, 10pJ) with low overhead
  - Keeps almost all (>99%) of data operations on chip (32GB/s, 100pJ)
- The Imagine processor demonstrates the advantages of streaming for image and signal processing
  - 9.6GFLOPs, 19.2GOPs, 6W - measured
- Stream processing is applicable to a wide range of applications
  - Scientific computing
  - Packet processing