Exploring Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators

Yunsup Lee¹, Rimas Avizienis¹, Alex Bishara¹, Richard Xia¹, Derek Lockhart², Christopher Batten², Krste Asanovic¹

¹The Parallel Computing Lab, UC Berkeley
²Computer Systems Lab, Cornell University
DLP Kernels Dominate Many Computational Workloads

- Graphics Rendering
- Computer Vision
- Audio Processing
- Physical Simulation
DLP Accelerators are Getting Popular

Sandy Bridge

Knights Ferry

Fermi

Tegra
Important Metrics

- Area Efficiency: Performance / Area
- Energy Efficiency: Energy / Task
- Programmability (How hard is it to write code?)
Efficiency vs. Programmability

- Vector
- MIMD
- DLP
Efficiency vs. Programmability

- Maven/Vector-Thread
- Vector
- MIMD
- DLP
Efficiency vs. Programmability

- Maven/Vector-Thread
- Vector
- MIMD
- DLP
- Irregular DLP
Efficiency vs. Programmability

Programmability vs. Efficiency

- Maven/Vector-Thread
- Vector
- MIMD

DLP vs. Irregular DLP
Efficiency vs. Programmability

- Maven/Vector-Thread
- Vector
- GPU
- SIMT
- MIMD

DLP

Irregular DLP
Outline

- Data-Parallel Architecture
- Design Patterns
  - MIMD, Vector-SIMD, Subword-SIMD, SIMT, Maven/Vector-Thread
- Microarchitectural Components
- Evaluation Framework
- Evaluation Results
DLP Pattern #1: MIMD

Programmer’s Logical View

Memory

Architectural Registers
DLP Pattern #1: MIMD

Programmer’s Logical View

Memory

Architectural Registers
DLP Pattern #1: MIMD

Vector-Vector Add

\[
\text{loop:} \\
\text{load} \ a, \ a\_ptr \\
\text{load} \ b, \ b\_ptr \\
\text{add} \ c, \ a, \ b \\
\text{store} \ c, \ c\_ptr \\
a\_ptr++ \\
b\_ptr++ \\
c\_ptr++ \\
\text{branch}
\]

Masked Filter

\[
\text{loop:} \\
\text{load} \ a, \ a\_ptr \\
a\_ptr++ \\
\text{branch} \ a = 0 \\
\text{op0} \\
\text{op1} \} \ \text{FILTER OP} \\
\ldots \\
\text{branch}
\]

Programmer’s Logical View

![Diagram showing the logical view of the vector-vector add and masked filter operations. The diagram includes a memory block and processors labeled HT, μT0, μT1, μT2, μT3, μT4, and μTi, connected by arrows indicating data flow.]
DLP Pattern #1: MIMD

Typical Microarchitecture

Instr Memory

\[ \mu T_0 \]
\[ \mu T_1 \]

\[ \mu T_2 \]
\[ \mu T_3 \]

Multi-threaded Cores

Data Memory

Examples: Tilera, Rigel (both single threaded)

Programmer’s Logical View
DLP Pattern #2: Vector-SIMD

Programmer’s Logical View

Vector-SIMD Arithmetic Instructions

Vector-SIMD Memory Instructions

Architectural Vector Register with 4 Elements
DLP Pattern #2: Vector-SIMD

**Vector-Vector Add**

```
loop:
  vload A, a_ptr
  vload B, b_ptr
  vadd C, A, B
  vstore C, c_ptr
  a_ptr++
  b_ptr++
  c_ptr++
  branch
```

**Masked Filter**

```
loop:
  vload A, a_ptr
  vset F, A = 0
  vop0 under flag F
  vop1 under flag F
  ...
  a_ptr++
  branch
```

*Programmer’s Logical View*
DLP Pattern #2: Vector-SIMD

Typical Microarchitecture

Transmission Memory

CP

VIU

μT0 μT2

Vector
Lanes

μT1 μT3

VMU

Data Memory

Examples: Crays, NEC SX, T0, VIRAM

Programmer’s Logical View

HT

μT0 μT1 μT2 μT3

CT0

μTi

CTj

Memory
DLP Pattern #3: Subword-SIMD

Typical Microarchitecture

Examples:
- Intel SSE/AVX
- ARM Neon
- PowerPC Altivec

Programmer’s Logical View
Programmer’s Logical View

Microthread Block w/ 4 Microthreads

HT

μT0

μT1

μT2

μT3

μT4

μTi

Memory
DLP Pattern #4: SIMT

Vector-Vector Add

\[
\begin{align*}
    a_{\text{ptr}} &= a_{\text{base}} \times \text{thread}_{\text{id}} \\
    b_{\text{ptr}} &= b_{\text{base}} \times \text{thread}_{\text{id}} \\
    c_{\text{ptr}} &= c_{\text{base}} \times \text{thread}_{\text{id}}
\end{align*}
\]

**Vector-Vector Add**

\[
\begin{align*}
    a_{\text{ptr}} &= a_{\text{base}} \times \text{thread}_{\text{id}} \\
    b_{\text{ptr}} &= b_{\text{base}} \times \text{thread}_{\text{id}} \\
    c_{\text{ptr}} &= c_{\text{base}} \times \text{thread}_{\text{id}}
\end{align*}
\]

**Programmer’s Logical View**

- **load** \( a, a_{\text{ptr}} \)
- **load** \( b, b_{\text{ptr}} \)
- **add** \( c, a, b \)
- **store** \( c, c_{\text{ptr}} \)
DLP Pattern #4: SIMT

Typical Microarchitecture

Instr Memory

VIU

μT0
μT2

μT1
μT3

VMU

Data Memory

Vector Lanes

Examples:
NVIDIA Fermi

Programmer’s Logical View

HT
μT0 μT1 μT2 μT3 μT4
μTi

Memory
AMD GPU Architectures?

- Would like to learn what exactly is in AMD architectures
- In conventional architecture terms
DLP Pattern #5: Vector-Thread (VT)

Programmer’s Logical View

Vector and Scalar Control Flow Instructions

Vector and Scalar Memory Instructions
DLP Pattern #5: Vector-Thread (VT)

Vector-Vector Add

loop:
  vload A, a_ptr
  vload B, b_ptr
  vfetch ut_code
  vstore C, c_ptr
  a_ptr++
  b_ptr++
  c_ptr++
  branch

ut_code:
  add c, a, b
  stop

Vector-Vector Add

loop:
  vload A, a_ptr
  vload B, b_ptr
  vfetch ut_code
  vstore C, c_ptr
  a_ptr++
  b_ptr++
  c_ptr++
  branch

ut_code:
  add c, a, b
  stop

Programmer’s Logical View
DLP Pattern #5: Vector-Thread (VT)

Maven Microarchitecture

Instruction Memory

CP

VU

μT0
μT2
μT1
μT3

VMU

Data Memory

Vector Lanes

Early example: MIT Scale

Programmer’s Logical View

HT
CT0
μT0
μT1
μT2
μT3
CTj
μTi

Memory
What’s interesting about Maven?

- Simple microarchitecture, very similar to traditional vector machines
  - Earlier Scale processor introduced vector-thread programming model but had much more complex design
- Same or better efficiency on regular DLP codes as traditional vector machine
- Easier to program and/or higher efficiency on irregular DLP codes
- Uses same ISA for scalar and vector unit
  - Significantly reduced compiler/library development effort
- Should be more efficient than pure SIMT-style machines
Outline

- Data Parallel Architectural Design Patterns
- Microarchitectural Components
- Evaluation Framework
- Evaluation Results
Focus on the Tile
uArchitecture

- Developed a library of parameterized synthesizable RTL components
Retimable Long-latency Functional Units

- 32-bit integer multiplier, divider
- Single-precision add, multiply, divide, square root
5-stage Multi-threaded Scalar Core

- Change number of entries in register file (32, 64, 128, 256) to vary degree of multi-threading (1, 2, 4, 8 threads)
Vector Lanes

- Vector registers and ALUs
- Density-time Execution
- Replicate the lanes and execute in lock step for higher throughput
- Flag Registers
**Density-Time Execution**

**Simple Implementation**
- execute all $N$ operations, turn off result writeback according to mask

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Density-Time Implementation**
- scan mask vector and only execute elements with non-zero masks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$C[4]$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$C[5]$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$Write$ $data$ $port$

$Write$ $Enable$

$Write$ $data$ $port$
Vector Issue Unit

- Vector-SIMD: VIU only handles issue, data-dependent control in software via flag registers
- Maven: VIU fetches instructions, PVFB handles uT branches in hardware
Vector-Fetched Branches

loop:
  vload A, a_ptr
  vfextract ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
vector-fetched branches

loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
Vector-Fetched Branches

loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
Vector-Fetched Branches

loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
Vector-Fetched Branches

loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
Vector-Fetched Branches

**Algorithm**

```
loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop
```
loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

ut_code:
  branch a = 0
  op0
  op1
  ...
  stop

VF Branch Tradeoffs
- Simple & keeps μTs coherent even after branch
- Large branch resolution latency
- Can data- and clock-gate inactive μTs
Density-Time Tradeoffs

- More complicated but higher performance
- Completely avoids any work for inactive μTs
Pending Vector-Fragment Buffer (PVFB)

- Vector fragment contains < PC, mask of active uTs >
- At taken branch, if uTs diverge, push new fragment representing uTs taking branch into PVFB
  - New fragment merges with existing fragments with same PC
  - PVFB must be sized to allow one fragment entry per uT
    - Worst case, every uT is in separate fragment
- Keep executing uTs on fall-through path
- When hit “stop” on current fragment, pop next fragment from PVFB
  - Multiple possible policies on which fragment to execute next
- When PVFB empty, execute next vector-fetch
  - Vector-thread ISA semantics are that all effects of a vector-fetch are visible before next vector-fetch
PVFB holds Pending Vector Fragments (PC + uT mask)

```
ut_code:
  0x10c: op0
  0x110: branch, 0x200
  0x114: op1
  0x118: op2
  ...
  0x200: op64
```
PVFB holds Pending Vector Fragments (PC + uT mask)

`ut_code:`

- 0x10c: op0
- 0x110: branch, 0x200
- 0x114: op1
- 0x118: op2
- ...
- 0x200: op64

(PC, uT mask)

(0x110, 1111)
PVFB holds Pending Vector Fragments (PC + uT mask)

**ut_code:**
0x10c: op0
0x110: branch, 0x200 (0x114, 0101) (0x200, 1010)
0x114: op1
0x118: op2
...
0x200: op64
PVFB holds Pending Vector Fragments (PC + uT mask)

**ut_code:**
- 0x10c: op0
- 0x110: branch, 0x200 (0x114, 0101) (0x200, 1010)
- 0x114: op1
- 0x118: op2
- ...
- 0x200: op64

Execute
PVFB holds Pending Vector Fragments (PC + uT mask)

\texttt{ut\_code:}

- 0x10c: op0
- 0x110: branch, 0x200 (0x114, 0101) (0x200, 1010)
- 0x114: op1
- 0x118: op2
- ...
- 0x200: op64

Execute

Put this vector fragment into PVFB
Want to maximize opportunities to dynamically merge fragments in PVFB.

Challenge for Maven is that there is no ISA support to indicate when to converge
  - Adding this makes ISA messy (can wayward uTs lock up machine?)

Policy boils down to which fragment to execute next.

FIFO – oblivious scheme

1-stack – keep PCs in sorted order, execute lowest PC fragment next (from Aamodt)

2-stack – put backwards branches on second sorted stack
- VMU Handles unit stride, constant stride memory operations
- Vector-SIMD: VMU handles scatter, gather
- Maven: VMU handles uT loads and stores
Blocking Caches

- Access Port Width
- Refill Port Width
- Cache Line Size
- Total Capacity
- Associativity
Non-Blocking Caches

- Access Port Width
- Refill Port Width
- Cache Line Size
- Total Capacity
- Associativity
- # MSHR
- # secondary misses per MSHR
Design Space Exploration for Microarchitectural components

- Number of entries in scalar register file
  - 32, 64, 128, 256 (1, 2, 4, 8 threads)
- Number of entries in vector register file
  - 32, 64, 128, 256
- Architecture of vector register file
  - 6r3w unified register file, 4x 2r1w banked register file
- Per-bank integer ALUs
- Density time execution
- PVFB schemes
  - FIFO, 1-stack, 2-stack
Five Tile Configurations

- **MIMD Tile**
  - I$-to-Network Xbar
  - Load/Store Xbar
  - D$-to-Network Xbar

- **Multi-core Vector-SIMD Tile**
  - I$-to-Network Xbar
  - Load/Store Xbar
  - D$-to-Network Xbar

- **Multi-core Maven Tile**
  - I$-to-Network Xbar
  - Load/Store Xbar
  - D$-to-Network Xbar

- **Multi-lane Vector-SIMD Tile**
  - I$-to-Network Xbar
  - Load/Store Xbar
  - D$-to-Network Xbar

- **Multi-lane Maven Tile**
  - I$-to-Network Xbar
  - Load/Store Xbar
  - D$-to-Network Xbar
MIMD
Vector-SIMD

Multi-lane Tile

Control Processor
- CP Regfile
  - 32x32b
  - 2r2w
- CP Embedding Queues
- Vector Issue Unit
  - Fetch/Decode
  - Issue
  - PVFB
  - PC
  - Mask
- VMU μT Queue
- VMU Vector Queue

Vector Lane
- VAU0 Sequencer
- VAU1 Sequencer
- VLU Sequencer
- VSU Sequencer
- VGU Sequencer
- Flag

Vector Memory Unit
- μTAQ
- μTLDQ
- μTSDQ
- VLDQ
- VSDQ

Data Cache Req & Resp Arbiters and Crossbars
- VLAGU
- VSAGU

CP Instr Cache
- 16 KB

VT Instr Cache
- 2 KB

Shared Data Cache
- 64 KB

Port to Main Memory
Vector-Thread

Multi-lane Tile

Multi-core Single-lane tile

Control Processor
CP Regfile
32x32b
2r2w
CP Embedding Queues
Vector Issue Unit
Fetch/Decode
PVFB
PC
Mask
Vector Lane
VAU0 Sequencer
VAU1 Sequencer
VLU Sequencer
VSU Sequencer
VGU Sequencer
Vector Regfile (128x32b 6r3w)

Load/Store Xbar

D$-to-Network Xbar

Load/Store Xbar

D$-to-Network Xbar

CP Instr Cache
16 KB

VT Instr Cache
2 KB

Shared Data Cache
64 KB

Port to Main Memory

Data Cache Req & Resp
Arbiters and Crossbars

\[\text{I$} \rightarrow \text{D$} \rightarrow \text{Vector} \rightarrow \text{CP} \rightarrow \text{Load/Store} \rightarrow \text{Network}\]

\[\text{I$} \rightarrow \text{D$} \rightarrow \text{Vector} \rightarrow \text{CP} \rightarrow \text{Load/Store} \rightarrow \text{Network}\]
Vector-Thread

Multi-lane Tile

Multi-core Single-lane tile

Control Processor
CP Regfile
32x32b 2r2w
int
CP Embedding Queues

VAU0 Sequencer
VAU1 Sequencer
VLU Sequencer
VSU Sequencer
VGU Sequencer

Vector Issue Unit
Fetch/Decode
PVFB
PC
Mask

Vector Lane

Load/Store Xbar
D$ D$ D$ D$
D$-to-Network Xbar

Load/Store Xbar
I$ I$ I$ I$
I$-to-Network Xbar

D$ D$ D$ D$
D$-to-Network Xbar

Data Cache Req & Resp
Arbiters and Crossbars

CP Instr Cache
16 KB

VT Instr Cache
2 KB

Shared Data Cache
64 KB

Port to Main Memory
Outline

- Data Parallel Architectural Design Patterns
- Microarchitectural Components
- Evaluation Framework
- Evaluation Results
Programming Methodology

- **MIMD**
  - GCC C/C++ Cross Compiler
  - Custom lightweight threading library
  - Applications explicitly manage thread scheduling

- **Vector-SIMD**
  - Leverage built-in GCC vectorizer for mapping very simple regular DLP code
  - Use GCC’s inline assembly extensions for more complicated code

- **Maven**
  - Use C++ Macros with libvt (special library, which glues control thread and microthreads)
## Microbenchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>vvadd</td>
<td>1000 element FP vector-vector add</td>
<td></td>
<td></td>
<td></td>
<td>100.0</td>
</tr>
<tr>
<td>bsearch</td>
<td>1000 look-ups into a sorted array</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>inner-loop rewritten with cond. mov</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

## Application Kernels

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>viterbi</td>
<td>Decode frames using Viterbi alg.</td>
<td></td>
<td></td>
<td></td>
<td>100.0</td>
</tr>
<tr>
<td>rsort</td>
<td>Radix sort on an array of integers</td>
<td></td>
<td></td>
<td></td>
<td>100.0</td>
</tr>
<tr>
<td>kmeans</td>
<td>K-means clustering algorithm</td>
<td></td>
<td></td>
<td></td>
<td>100.0</td>
</tr>
<tr>
<td>dither</td>
<td>Floyd-Steinberg dithering</td>
<td>0.2</td>
<td>0.4</td>
<td>0.7</td>
<td>98.7</td>
</tr>
<tr>
<td>physics</td>
<td>Newtonian physics simulation</td>
<td>6.9</td>
<td>15.0</td>
<td>28.7</td>
<td>49.3</td>
</tr>
<tr>
<td>strsearch</td>
<td>Knuth-Morris-Pratt algorithm</td>
<td>57.5</td>
<td>25.5</td>
<td>16.9</td>
<td>0.1</td>
</tr>
</tbody>
</table>
void vvadd_vt( int dest[], int src0[],
              int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );

        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load( &src0[i] );
        vsrc1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsr0,vsrc1),
                   ({{
                        vdest = vsr0 + vsrc1;
                    }}));

        vdest.store( &dest[i] );
    }
}
void vvaddvt(int dest[], int src0[], int src1[], int size)
{
    int vlen = vt::set_vlen(size);
    for (int i = 0; i < size; i += vlen)
    {
        vlen = vt::set_vlen(size - i);

        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load(&src0[i]);
        vsrc1.load(&src1[i]);

        vt::HardwareVector<int> vdest;

        VT_VFETCH((vdest), (vsr0,vsrc1),
        {
            vdest = vsr0 + vsrc1;
        });

        vdest.store(&dest[i]);
    }
}
```c++
void vvadd_vt( int dest[], int src0[],
        int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );
        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load( &src0[i] );
        vsrc1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsr0,vsrc1),
        ({{
            vdest = vsrc0 + vsrc1;
        }}));

        vdest.store( &dest[i] );
    }
}
```

Compiles to instruction which abstracts hardware vector length

C++ class that uses compiler vector types to enable automatic vector register allocation
void vvadd_vt( int dest[], int src0[],
               int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );
        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load( &src0[i] );
        vsrc1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsrc0,vsrc1),
          (vdest = vsrc0 + vsrc1; ));

        vdest.store( &dest[i] );
    }
}
void vvadd_vt( int dest[], int src0[], int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );
        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load( &src0[i] );
        vsrc1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsrc0,vsrcl),
        ({
            vdest = vsrc0 + vsrcl;
        }));

        vdest.store( &dest[i] );
    }
}
void vvadd_vt( int dest[], int src0[],
              int src1[], int size )
{
  int vlen = vt::set_vlen( size );
  for ( int i = 0; i < size; i += vlen )
  {
    vlen = vt::set_vlen( size - i );
    vt::HardwareVector<int> vsrc0, vsrc1;
    vsrc0.load( &src0[i] );
    vsrc1.load( &src1[i] );
    vt::HardwareVector<int> vdest;
    VT_VFETCH( (vdest), (vsr0,vsrc1),
               {{
                 vdest = vsrc0 + vsrc1;
               }});
    vdest.store( &dest[i] );
  }
}
```c++
void vvadd_vt( int dest[], int src0[],
               int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );

        vt::HardwareVector<int> vsrc0, vsrcl;
        vsrc0.load( &src0[i] );
        vsrcl.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsrcl, vsrcl),
                   {
                       vdest = vsrc0 + vsrcl;
                       if ( vdest < 0 )
                           error(vdest);
                       ...
                   });
}
```

Standard C++ allowed inside vector-fetched block including objects, conditionals, loops, and function calls

Currently system calls and exceptions not supported
Three Example Layouts

- **MIMD Tile**
- **Multi-Lane Maven Tile**
- **Multi-Core Maven Tile**
Need Gate-level Activity for Accurate Energy Numbers

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Post Place&amp;Route Statistical (mW)</th>
<th>Simulated Gate-level Activity (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMD 1</td>
<td>149</td>
<td>137-181</td>
</tr>
<tr>
<td>MIMD 2</td>
<td>216</td>
<td>130-247</td>
</tr>
<tr>
<td>MIMD 3</td>
<td>242</td>
<td>124-261</td>
</tr>
<tr>
<td>MIMD 4</td>
<td>299</td>
<td>221-298</td>
</tr>
<tr>
<td>Multi-core Vector-SIMD</td>
<td>396</td>
<td>213-331</td>
</tr>
<tr>
<td>Multi-lane Vector-SIMD</td>
<td>224</td>
<td>137-252</td>
</tr>
<tr>
<td>Multi-core Vector-Thread 1</td>
<td>428</td>
<td>162-318</td>
</tr>
<tr>
<td>Multi-core Vector-Thread 2</td>
<td>404</td>
<td>147-271</td>
</tr>
<tr>
<td>Multi-core Vector-Thread 3</td>
<td>445</td>
<td>172-298</td>
</tr>
<tr>
<td>Multi-core Vector-Thread 4</td>
<td>409</td>
<td>225-304</td>
</tr>
<tr>
<td>Multi-core Vector-Thread 5</td>
<td>410</td>
<td>168-300</td>
</tr>
<tr>
<td>Multi-lane Vector-Thread 1</td>
<td>205</td>
<td>111-167</td>
</tr>
<tr>
<td>Multi-lane Vector-Thread 2</td>
<td>223</td>
<td>118-173</td>
</tr>
</tbody>
</table>
Outline

- Data Parallel Architectural Design Patterns
- Microarchitectural Components
- Evaluation Framework
- Evaluation Results
Results running `bsearch-cmv`
Results running `bsearch-cmv`

- Faster
- Lower Energy
Results running `bsearch-cmv`
Results running \textit{bsearch-cmv}
Results running `bsearch-cmv`
6r3w Vector Register File is Area Inefficient
6r3w Vector Register File is Area Inefficient

- BERKELEY PAR LAB
- MIMD Tile
- Vector-Thread Tile

Diagram showing normalized area for different tile sizes (r32, r64, r128, r256) compared to a base of 1.00.
Banked Vector Register File

Vector Lane

- VAU0 Sequencer
- VAU1 Sequencer
- VLU Sequencer
- VSU Sequencer
- VGU Sequencer

Vector Regfile (128x32b 6r3w)

Banked Vector Regfile

- Bank 0
  - 32x32b 2r1w
- Bank 1
  - 32x32b 2r1w
- Bank 2
  - 32x32b 2r1w
- Bank 3
  - 32x32b 2r1w
Results running \textit{bsearch-cmv}
Add Per-Bank Integer ALUs
Results running bsearch-cmv
Banked Vector Register File with Per-Bank Integer ALUs

Vector-Thread Tile

Banking

Local ALUs

Normalized Area

MIMD Tile
Bank Vector Register File
Per Bank Integer ALUs

- ctrl
- reg
- mem
- fp
- int
- cp
- i$
- d$

MIMD Tile

Vector-Thread Tile

Banking

Local ALUs

Normalized Area

r32, r64, r128, r256
Bank Vector Register File
Per Bank Integer ALUs

Vector-Thread Tile
Banking
Local ALUs

MIMD Tile

Normalized Area

ctrl  reg  mem  int  cp  i$  d$  fp
Bank Vector Register File
Per Bank Integer ALUs

- BERKELEY PAR LAB
- Bank Vector Register File
- Per Bank Integer ALUs

- Vector-Thread Tile
- Banking
- Local ALUs

- ctrl
- reg
- mem
- int
- cp
- i$
- fp
- d$

Normalized Area

- MIMD Tile
- r32, r64, r128, r256

- Local ALUs
- r32, r64, r128, r256, r128+b, r256+b, r256+bi
Result of Design Space Exploration:
256 Registers Per Lane
Banked Vector Register File
Add Local Integer ALUs
Results running \textit{bsearch} compared to \textit{bsearch-cmv}
Results running `bsearch` compared to `bsearch-cmv`

- 13.5x Faster
- 9x Less Energy
Results running *bsearch* compared to *bsearch-cmv*

<table>
<thead>
<tr>
<th>Name</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>
Results running `bsearch` compared to `bsearch-cmv`
Results running `bsearch` compared to `bsearch-cmv`

<table>
<thead>
<tr>
<th>Name</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch 1-s</td>
<td>23.8</td>
<td>23.4</td>
<td>11.7</td>
<td>41.0</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

Active uT Distribution (%)
Results running *bsearch* compared to *bsearch-cmv*

<table>
<thead>
<tr>
<th>Name</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch 1-s</td>
<td>23.8</td>
<td>23.4</td>
<td>11.7</td>
<td>41.0</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

- FIFO
- FIFO+dt
- 1-stack
- 1-stack+dt
- cmv +FIFO

Normalized Energy / Task vs. Normalized Tasks / Sec

Active uT Distribution (%)
Results running \textit{bsearch} compared to \textit{bsearch-cmv}

<table>
<thead>
<tr>
<th>Name</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch 1-s</td>
<td>23.8</td>
<td>23.4</td>
<td>11.7</td>
<td>41.0</td>
</tr>
<tr>
<td>bsearch 2-s</td>
<td>10.1</td>
<td>26.8</td>
<td>49.2</td>
<td>13.9</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

Active uT Distribution (%)
Results running `bsearch` compared to `bsearch-cmv`
Results running `bsearch` compared to `bsearch-cmv`

<table>
<thead>
<tr>
<th>Name</th>
<th>1-25</th>
<th>26-50</th>
<th>51-75</th>
<th>76-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>77.6</td>
<td>12.4</td>
<td>5.1</td>
<td>4.8</td>
</tr>
<tr>
<td>bsearch 1-s</td>
<td>23.8</td>
<td>23.4</td>
<td>11.7</td>
<td>41.0</td>
</tr>
<tr>
<td>bsearch 2-s</td>
<td>10.1</td>
<td>26.8</td>
<td>49.2</td>
<td>13.9</td>
</tr>
<tr>
<td>bsearch-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

Normalized Energy / Task

Normalized Tasks / Sec

Active uT Distribution (%)
Area Overhead of Convergence and Density-Time is Negligible

MIMD Tile

Density Time

FIFO

1S

2S

Normalized Area

ctrl
reg
mem
fp
time
int
cp
i$
d$
Area Overhead of Convergence and Density-Time is Negligible

Fix Design Parameters:
2-Stack PVFB
Density-Time Execution
Results Running Application Kernels

**Performance**

- **viterbi**
- **rsort**
- **kmeans**
- **dither**
- **physics**
- **strsearch**

Normalized Tasks / Second

Performance / Area

Normalized Tasks / Second / Area
Results Running Application Kernels

Normalized Tasks / Second

More Irregular

Normalized Energy / Task

Normalized Tasks / Second / Area

Tasks: viterbi, rsort, kmeans, dither, physics, strsearch
Multi-threading is not Effective on DLP Code

Normalized Energy / Task vs. Normalized Tasks / Second

- viterbi
- rsort
- kmeans
- dither
- physics
- strsearch

Normalized Energy / Task vs. Normalized Tasks / Second / Area

- r32
Vector-SIMD is Faster and/or More Efficient than MIMD

- **viterbi**, **rsort**, **kmeans**, **dither**, **physics**, **strsearch**

Normalized Tasks / Second

Normalized Energy / Task
Vector-Thread is More Efficient than Vector-SIMD

Normalized Tasks / Second

viterbi | rsort | kmeans | dither | physics | strsearch

Normalized Tasks / Area

r32 | mlane

Normalized Energy / Task

r32 | mlane

Effective tasks per second and area are normalized to the baseline.
Multi-Lane Tiles are More Efficient than Multi-Core Tiles

- viterbi
- rsort
- kmeans
- dither
- physics
- strsearch

Normalized Tasks / Second

Normalized Energy / Task

Normalized Tasks / Second / Area
Results running \textit{vvadd}

![Graph showing normalized energy per task versus normalized tasks per second. The point labeled \textit{vec ld/st} is highlighted at the bottom right.]
uT Memory Accesses Limits
Access-Execute Decoupling

9x Slower
5x More Energy
Memory Coalescing Helps, but Far Behind Vector Instructions
28nm Vector-Thread Test Chip

- A follow-on to the Maven vector-thread architecture
- Taped out May 29th, 2011
- Stay tuned!
Conclusion

- Vector architectures are more area and energy efficient than MIMD architectures.
- The Maven vector-thread architecture is superior to traditional vector-SIMD architectures, by providing both greater efficiency and easier programmability.

- This work was supported in part by Microsoft (Award #024263) and Intel (Award #024894, equipment donations) funding and by matching funding from U.C. Discovery (Award #DIG07-10227).