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
Comparing DLP Accelerator Architectures

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

Diagram showing the comparison between different architectures:
- **Maven/Vector-Thread**: High efficiency and high programmability.
- **Vector**: High efficiency and medium programmability.
- **MIMD**: Medium efficiency and high programmability.
- **DLP**: Low efficiency and low programmability.
- **Irregular DLP**: Low efficiency and high programmability.
Efficiency vs. Programmability

- MIMD
- Vector
- Maven/Vector-Thread

DLP

Irregular DLP
Efficiency vs. Programmability

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

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

Efficiency

Programmability

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
Vector-Vector Add

```
loop:
  load  a, a_ptr
  load  b, b_ptr
  add   c, a, b
  store c, c_ptr
  a_ptr++
  b_ptr++
  c_ptr++
  branch
```

Masked Filter

```
loop:
  load  a, a_ptr
  a_ptr++
  branch a = 0
  op0
  op1
  FILTER OP
  ...
  branch
```
Typical Microarchitecture

Instr Memory

μT0
μT1

μT2
μT3

Data Memory

Multi-threaded Cores

Examples: Tilera, Rigel (both single threaded)

Programmer’s Logical View

HT μT0 μT1 μT2 μT3 μT4 μTi

Memory
DLP Pattern #2: Vector-SIMD

Programmer’s Logical View
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**

Diagram showing the interaction between data structures and processors.
DLP Pattern #2: Vector-SIMD

Typical Microarchitecture

Examples: Crays, NEC SX, T0, VIRAM

Programmer’s Logical View
DLP Pattern #3: Subword-SIMD

Typical Microarchitecture

Instruction Memory

CP

VIU

SIMD Unit

Data Memory

VMU

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

Programmer’s Logical View

HT

CT0

CTj

SIMD Unit

Memory
DLP Pattern #4: SIMT

Programmer’s Logical View

Microthread Block w/ 4 Microthreads

Memory
DLP Pattern #4: SIMT

Vector-Vector Add

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

**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

Examples: NVIDIA Fermi

Programmer’s Logical View
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

Programmer’s Logical View

Memory

HT

CT0

µT0

µT1

µT2

µT3

CTj

µTi
DLP Pattern #5: Vector-Thread (VT)

Maven Microarchitecture

Instruction Memory

CP

VT0

VT1

VT2

VT3

VU

VMU

Data Memory

Early example: MIT Scale

Programmer’s Logical View

Memory

HT

CT0

µT0

µT1

µT2

µT3

CTj

µT

µTi
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

Tile

On-Chip Network

Outer-Level Memory System

Mem Ctrl

Mem Ctrl

Mem Ctrl
Developed a library of parameterized synthesizable RTL components
- 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

\[
\begin{align*}
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0
\end{align*}
\]

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

\[
\begin{align*}
M[7] &= 1 \\
M[6] &= 0 \\
M[5] &= 1 \\
M[4] &= 1 \\
M[3] &= 0 \\
M[2] &= 0 \\
M[1] &= 1 \\
M[0] &= 0
\end{align*}
\]


\[C[5] \quad C[4] \]

\[C[1] \]

\[Write data port\]
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
  - vfetch ut_code
  - a_ptr++
  - branch

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

loop:
\begin{verbatim}
  vload A, a_ptr
  vfitch ut_code
  a_ptr++
  branch
\end{verbatim}

\texttt{ut\_code:}
\begin{verbatim}
  branch a = 0
  op0
  op1
  ...
  stop
\end{verbatim}
loop:
\texttt{vload A, a_ptr}
\texttt{vfetch ut_code}
\texttt{a_ptr++}
\texttt{branch}
\texttt{ut_code: branch a = 0}
\texttt{op0}
\texttt{op1}
\texttt{...}
\texttt{stop}
loop:
  vload A, a_ptr
  vfetch ut_code
  a_ptr++
  branch

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

```plaintext
vload A, a_ptr
vfetch ut_code
a_ptr++
branch
```

`ut_code:`

```plaintext
branch a = 0
op0
op1
...
stop
```
Vector-Fetched Branches

loop:
  vload A, a_ptr
  vfetched 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
loop:
  vload A, a_ptr
  vf fetch ut_code
  a_ptr++
  branch

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

Vector-Fetched Branches

Diagram showing the flow of operations in a parallel processing system, with annotations for vload, vf fetch, branch, and associated memory operations.
Vector-Fetched Branches

VF Branch Tradeoffs

- Simple & keeps μTs coherent even after branch
- Large branch resolution latency
- Can data- and clock-gate inactive μTs

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

ut_code:
branch a = 0
op0
op1
...
stop
```
Density-Time Tradeoffs

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

- Vector fragment contains $<\text{PC}, \text{mask of active uTs}>$
- 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
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: (PC, uT mask)
  0x10c: op0
  0x110: branch, 0x200 (0x114, 0101) (0x200, 1010)
  0x114: op1
  0x118: op2
  ...
  0x200: op64
```
PVFB holds Pending Vector Fragments (PC + uT mask)

\textbf{ut\_code:}

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

\textbf{(PC, uT mask)}

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

ut_code:

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

(PC, uT mask)

Execute

Put this vector fragment into PVFB
PVFB Management Schemes

- 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
Vector Memory Unit

- 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
Vector-SIMD

Multi-lane Tile

Control Processor

Vector Lane

Vector Issue Unit

Vector Memory Unit

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
Vector-Thread
Vector-Thread

Multi-lane Tile

Multi-core Single-lane tile

Control Processor
CP Regfile
32x32b
2r2w
CP Embedding Queues

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

Vector Regfile (128x32b 6r3w)

Vector Issue Unit
Fetch/Decode
PVFB
PC
Mask

VIU Queue
Branch Resolution

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

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

I$ 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), (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,vsrc1),
    {
      vdest = vsrc0 + vsrc1;
    } );

    vdest.store( &dest[i] );
  }
}
```cpp
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), (vsr0,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,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,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;
            if ( vdest < 0 )
                error(vdest);
            ...
        });
    }
}
Evalutation Methodology

Software Toolflow

- C++ Application
  - Native Compiler
    - Native Binary
  - Cross Compiler
    - Target Binary
    - Target ISA Sim

Hardware Toolflow

- Verilog RTL
  - Synthesis Place&Route
  - Gate-Level Model
    - Verilog Simulator
  - Layout
    - Switching Activity
      - Power Analysis

Results

- Area & Cycle Time
- Cycle Count
- Power
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*

![Graph showing energy consumption and normalized tasks per second for different task sizes and configurations. The graph includes a line graph and a bar chart, with axes labeled as follows: y-axis: Normalized Energy / Task, x-axis: Normalized Tasks / Sec. The graph compares performance metrics for different task sizes r32 and r64.]
Results running \textit{bsearch-cmv}

The graph shows the normalized energy per task on the y-axis against normalized tasks per second on the x-axis. The line labeled \textit{mimd-c4} indicates the energy performance for different task sizes (r32, r64, r128, r256).

The bar chart on the right compares energy consumption across various tasks and components, with different colors representing different categories: \texttt{ctrl}, \texttt{cp}, \texttt{reg}, \texttt{i$}, \texttt{mem}, \texttt{d$}, \texttt{fp}, \texttt{int}, and \texttt{leak}.
Results running *bsearch-cmv*
6r3w Vector Register File is Area Inefficient
6r3w Vector Register File is Area Inefficient

Vector-Thread Tile

MIMD Tile
Banked Vector Register File
Results running `bsearch-cmv`
Add Per-Bank Integer ALUs
Results running *bsearch-cmv*

- **Comparison Chart**
  - **Y-axis**: Normalized Energy / Task
  - **X-axis**: Normalized Tasks / Sec

- **Legend**:
  - `mimd-c4`
  - `vt-c4v1`
  - `vt-c4v1+b`
  - `vt-c4v1+bi`

- **Data Points**:
  - `r256`
  - `r128`
  - `r64`
  - `r32`
Banked Vector Register File with Per-Bank Integer ALUs

MIMD Tile

Vector-Thread Tile

Banking

Local ALUs

Normalized Area
### Bank Vector Register File

#### Per Bank Integer ALUs

<table>
<thead>
<tr>
<th>Bank Size</th>
<th>ctrl</th>
<th>reg</th>
<th>int</th>
<th>cp</th>
<th>mem</th>
<th>i$</th>
<th>d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>f32</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>f64</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>f128</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>f256</td>
<td>80</td>
<td>90</td>
<td>108</td>
<td>126</td>
<td>144</td>
<td>162</td>
<td>180</td>
</tr>
</tbody>
</table>

---

**Diagram Description:**

- The diagram illustrates the normalized area for MIMD and Vector-Thread Tile ALUs.
- Each bar represents the normalized area for different bank sizes.
- The legend at the top indicates the different components represented in the bars.
- The Vector-Thread Tile and Local ALUs are highlighted with specific colors.

---

**Legend:**

- ctrl: Control
- reg: Register
- int: Integer
- cp: Configuration
- mem: Memory
- i$: Instruction
- d$: Data
- fp: Floating Point
Bank Vector Register File
Per Bank Integer ALUs

Vector-Thread Tile

Banking

Local ALUs

MIMD Tile

Normalized Area

ctrl
reg
mem
fp
cp
int
i$
dl$

Sizes:

r32
r64
r128
r256
r128+b
r256+b
r128+bi
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 \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-cmv</td>
<td>1.0</td>
<td>3.3</td>
<td>5.8</td>
<td>89.9</td>
</tr>
</tbody>
</table>

Normalized Energy / Task vs. Normalized Tasks / Sec

- FIFO
- cmv + FIFO
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>

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

Active uT Distribution (%)

<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>

Normalized Energy / Task

Normalized Tasks / Sec
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>

- **Normalized Energy / Task**
- **Normalized Tasks / Sec**
- **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 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>
### 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>

#### Graph:
- **FIFO**: Points showing the energy distribution for FIFO.
- **FIFO+dt**: Points showing the energy distribution for FIFO with delay.
- **1-stack**: Points showing the energy distribution for 1-stack.
- **2-stack+dt**: Points showing the energy distribution for 2-stack with delay.
- **1-stack+dt**: Points showing the energy distribution for 1-stack with delay.
- **2-stack**: Points showing the energy distribution for 2-stack.
- **cmv + FIFO**: Points showing the energy distribution for cmv with FIFO.
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>
Area Overhead of Convergence and Density-Time is Negligible

MIMD Tile

Density Time

1S

2S

FIFO

Normalized Area

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

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

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

Normalized Tasks / Second

Normalized Tasks / Second / Area
Results Running Application Kernels

Performance

Normalized Tasks / Second

Performance / Area

Normalized Tasks / Second / Area
Results Running Application Kernels

More Irregular

Normalized Tasks / Second

Normalized Tasks / Second / Area

viterbi  rsort  kmeans  dither  physics  strsearch
Results Running Application Kernels

viterbi  rsort  kmeans  dither  physics  strsearch

Normalized Tasks / Second

Normalized Energy / Task

No Vector-SIMD Implementation

Only Vector-Thread Implementation
Multi-threading is not Effective on DLP Code

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

_normalized Tasks / Second

Normalized Energy / Task

Normalized Tasks / Second / Area

r32

r32

r32

r32

r32

r32

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

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

Normalized Tasks / Second

Normalized Tasks / Second / Area

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

Normalized Energy / Task vs. Normalized Tasks / Second

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

Normalized Energy / Task vs. Normalized Tasks / Second / Area
Multi-Lane Tiles are More Efficient than Multi-Core Tiles

viterbi  rsort  kmeans  dither  physics  strsearch

Normalized Energy / Task vs. Normalized Tasks / Second

Normalized Energy / Task vs. Normalized Tasks / Second / Area
Results running \textit{vvadd}
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).