Architecture and compiler interaction and optimization opportunities

or...

why does the ISA change so often!
why does the ISA change so often!

1 Introduction
   Optimizing levels, advantages/disadvantages
2 Branch optimizations
   Branch Prediction, Critical Branch Detection, Inlining, etc.
3 Loops
   Unrolling, software pipelining, loop fussion/fission, ...
4 Memory/Caches
   Indexing, blocking/padding, prefetching, ...
5 SIMD and Multimedia Extensions
   Vectorization,...
1 Introduction

Out-of-order brings a performance boost but...
There is a limit:
- ILP available in the programs
- Number of in-flight instructions

Is that the end of the road?
No, we can look at the bigger picture
Processor

(+) It knows the state on real-time
  - (+) Precise and adaptable branch prediction
  - (+) Can manage data dependencies not known at compile-time (i.e. Memory dependences)

(-) Restricted program scope
  - (-) Small (if any) knowledge of program structure
  - (-) Limited scope for instruction scheduling.

(-) Hardware complexity
  - (-) Complex: hits the cycle time
  - (-) Expensive: in area, verification and power cost.
Compiler

(+) Global view of the program
- (+) Better information on program structure and high-level parallelism (loops, functions, etc)
- (+) Potentially larger perspective on instruction scheduling.

(+ ) More sophisticated analysis possible
- (+) No limits on chip area or cycle time
- (-) Restrictions on the total compile time and memory usage
Programming

Translate an idea to a program

A good place to optimize:

- Most effective/efficient algorithm
  - Sequential Search / Binary Search
- Efficient memory management
  - Memory usage for programs and data
- Is it worth considering the memory hierarchy?
Compiler

Translates a HLL program into an assembly code (of a specific processor ISA)

Compiler steps:

- Intermediate language translation
  - Similar to the assembly language but more extensive

- Translation to assembly code (code generation)
  - What instruction does the ISA support?
    - What if the ISA does not have a divide instruction?
  - How many registers does the ISA have?
    - Usually, each variable is assigned a register. How many variables/registers can we hold at a time?
  - Does the processor support static branch prediction?
Cross-Level optimizations

What can we optimize that is CPU dependent?

- **Branches**
  - Branch Prediction, Critical Branch Detection, Inlining, etc

- **Loops**
  - Unrolling, *software pipelining*, loop fusion/fission, ...

- **Memory/Caches**
  - Referencing, blocking/padding, prefetching,...

- **SIMD and multimedia extensions**
  - Vectorization
Memory and Processor Performance
Gap is Increasing

- CPU performance doubles every 18 months
- Memory capacity doubles every 18 months
  - But, speed of memory increases at a much lower rate
  - Speed gap increases by about 50% per year
- A large portion of program execution time is spent waiting for memory accesses to complete
  - and this % is ever increasing
- Can the compiler do things to reduce this time?
Locality: The Why of Memory Hierarchies

- Real programs typically access data that is close in time and space to data that has been recently accessed, this is called locality of reference.
- **Spatial Locality**: there is a small difference in the addresses of different data.
- **Temporal Locality**: there is a small difference in time between accesses to the same data.
- These characteristics have lead to the structure of the typical modern memory hierarchy.

```
DO I = 1, N
  A(I) = ...
  SUM = SUM + ...
ENDDO
```

- spatial locality
- temporal locality
The Memory Hierarchy

- Number of Levels (Why?)
- Line Size (Why?)
- Associativity (Why?)
- Types of Misses...

<table>
<thead>
<tr>
<th>Block address</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>
What a compiler can do

- A compiler can restructure applications to uncover and enhance locality

- A compiler has two dimensions to work with:
  - can change the order of memory access (control flow)
  - can change the layout of memory (declarations)

- Changing the order of memory accesses
  - the most common method
  - loop transformations: loop interchange, loop tiling, loop fission/fusion, ...
  - unimodular transformations

- Changing the layout of memory
  - data transformations: array transpose, array padding
Data Storage in Memory

Data storage order is language dependent:
- **Fortran stores multi-dimensional arrays “column-wise”**

  ![Fortran array diagram](image)

  Accessing array elements in storage order greatly improves performance.

- **C stores multi-dimensional arrays “row-wise”**

  ![C array diagram](image)

  Accessing array elements in storage order greatly improves performance.
Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

\[
\begin{align*}
do & \ j = 1, n \\
do & \ i = 1, n \\
& \quad \ldots \ a(i,j) \ldots \\
end do \\
end do
\end{align*}
\]

\[
\begin{align*}
do & \ i = 1, n \\
do & \ j = 1, n \\
& \quad \ldots \ a(i,j) \ldots \\
end do \\
end do
\end{align*}
\]
Padding

Leave empty spaces between structures so they do not conflict in the cache.
Cache Blocking

storage/cache line order

access order

Blocking:

We do the work on this block first
then we move on to the second block
and a leftover part

Repeat this process for the remaining part

Out-of-Cache
In-cache
Blocking for Cache (tiling)

Blocking for cache:
- An optimization that applies to data sets that do not fit into the (2nd level) data cache
- A way to increase spatial locality of reference (i.e. exploit full cache lines)
- A way to increase temporal locality of reference (i.e. to improve data re-use)
- It is beneficial mostly with multi-dimensional arrays

```
DO I=1,N
    ... (I) ...
ENDDO
```

```
DO i1=1,N,nb
    DO I=i1,min(i1+nb-1,N)
        ... (I) ...
    ENDDO
ENDDO
```

- `LNO:blocking=[on|off]` (default on)
- `LNO:blocking_size=n1,n2` (for L1 and L2)

By default L1=32KB and L2=1MB
Use `-LNO:cs2=8M` to specify the 8MB L2 cache

The inner loop is traversed only in the range of nb at a time
## Blocking: Example

The following loop nest:

```c
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    x[i][j] = y[i] + z[j]
```

- `z[j]` is reused for each `i` iteration
- for large `n` the array `z` will not be reused from the cache

### Blocking the loops for cache:

```c
For(it=0; it<n; it += nb)
  for(jt=0; jt<m; jt += nb)
    for(i=it; i<min(jt+nb,n); i++)
      for(j=jt; j<min(jt+nb,m); j++)
        x[i][j] = y[i] + z[j]
```

- `nb` elements of `z` array will be brought in to the cache and reused `nb` times before moving on to the next tile
Software Prefetching

Some processors support software prefetch instructions
- hints that a certain location will be needed in the near future
- usually have no side effect
- might be dropped if they interfere with useful work

Has potentially bad side effects
- may use space in load/store queue
- you may prefetch something that evicts useful data
- you may prefetch too early, and prefetched data is evicted before it use

Unlike other techniques it does not try to remove misses, but instead tries to hide the latency of misses.
Software Prefetch Example:

```c
int a;
int d[100000][16];

int main()
{
    unsigned i,j,k;
    for (i=0;i<1000;i++) {
        for (j=0;j<100000;j++) {
            for (k=0;k<16;k++) {
                a = a + d[j][k];
            }
        }
    }
}
```

Original 41.4 s
Prefetch 27.5 s
5 Multimedia Instruction Sets: SIMD and Vector
What is Multimedia Processing?

**Desktop:**
- 3D graphics (games)
- Speech recognition (voice input)
- Video/audio decoding (mpeg-mp3 playback)

**Servers:**
- Video/audio encoding (video servers, IP telephony)
- Digital libraries and media mining (video servers)
- Computer animation, 3D modeling & rendering (movies)

**Embedded:**
- 3D graphics (game consoles)
- Video/audio decoding & encoding (set top boxes)
- Image processing (digital cameras)
- Signal processing (cellular phones)
The Need for Multimedia ISAs

Why aren’t general-purpose processors and ISAs sufficient for multimedia (despite Moore’s law)?

Performance
- A 1.2GHz Athlon can do MPEG-4 encoding at 6.4fps
- One 384Kbps W-CDMA channel requires 6.9 GOPS

Power consumption
- A 1.2GHz Athlon consumes ~60W
- Power consumption increases with clock frequency and complexity

Cost
- A 1.2GHz Athlon costs ~$62 to manufacture and has a list price of ~$600 (module)
- Cost increases with complexity, area, transistor count, power, etc
Example: MPEG Decoding

Input Stream

Parsing

Dequantization

IDCT

Block Reconstruction

RGB->YUV

Output to Screen

Load Breakdown

10%

20%

25%

30%

15%
Example: 3D Graphics

Geometry Pipe
- Display Lists
  - Transform
  - Lighting
- Setup
  - Rasterization
  - Anti-aliasing
  - Shading, fogging
  - Texture mapping
  - Alpha blending
  - Z-buffer
  - Clipping
  - Frame-buffer ops
- Output to Screen

Load Breakdown
- 10%
- 10%
- 35%
- 55%
Characteristics of Multimedia Apps (1)

- Requirement for real-time response
  - “Incorrect” result often preferred to slow result
  - Unpredictability can be bad (e.g. dynamic execution)

- Narrow data-types
  - Typical width of data in memory: 8 to 16 bits
  - Typical width of data during computation: 16 to 32 bits
  - 64-bit data types rarely needed
  - Fixed-point arithmetic often replaces floating-point

- Fine-grain (data) parallelism
  - Identical operation applied on streams of input data
  - Branches have high predictability
  - High instruction locality in small loops or kernels
Characteristics of Multimedia Apps (2)

- **Coarse-grain parallelism**
  - Most apps organized as a pipeline of functions
  - Multiple threads of execution can be used

- **Memory requirements**
  - High bandwidth requirements but can tolerate high latency
  - High spatial locality (predictable pattern) but low temporal locality
  - Cache bypassing and prefetching can be crucial
Examples of Media Functions

- Matrix transpose/multiply
- DCT/FFT
- Motion estimation
- Gamma correction
- Haar transform
- Median filter
- Separable convolution
- Viterbi decode
- Bit packing
- Galois-fields arithmetic
- ...

(3D graphics)
(Video, audio, communications)
(Video)
(3D graphics)
(Media mining)
(Image processing)
(Image processing)
(Communications, speech)
(Communications, cryptography)
(Communications, cryptography)
Approaches to Mediaprocessing

- General-purpose processors with SIMD extensions
- Vector Processors
  - VLIW with SIMD extensions (aka mediaprocessors)
- DSPs
- ASICs/FPGAs

Multimedia Processing
SIMD Extensions for GPP

Motivation
- Low media-processing performance of GPPs
- Cost and lack of flexibility of specialized ASICs for graphics/video
- Underutilized datapaths and registers

Basic idea: sub-word parallelism
- Treat a 64-bit register as a vector of 2 32-bit or 4 16-bit or 8 8-bit values (short vectors)
- Partition 64-bit datapaths to handle multiple narrow operations in parallel

Initial constraints
- No additional architecture state (registers)
- No additional exceptions
- Minimum area overhead
# Overview of SIMD Extensions

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Extension</th>
<th>Year</th>
<th># Instr</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP</td>
<td>MAX-1 and 2</td>
<td>94,95</td>
<td>9,8 (int)</td>
<td>Int 32x64b</td>
</tr>
<tr>
<td>Sun</td>
<td>VIS</td>
<td>95</td>
<td>121 (int)</td>
<td>FP 32x64b</td>
</tr>
<tr>
<td>Intel</td>
<td>MMX</td>
<td>97</td>
<td>57 (int)</td>
<td>FP 8x64b</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow!</td>
<td>98</td>
<td>21 (fp)</td>
<td>FP 8x64b</td>
</tr>
<tr>
<td>Motorola</td>
<td>Altivec</td>
<td>98</td>
<td>162 (int,fp)</td>
<td>32x128b (new)</td>
</tr>
<tr>
<td>Intel</td>
<td>SSE</td>
<td>98</td>
<td>70 (fp)</td>
<td>8x128b (new)</td>
</tr>
<tr>
<td>MIPS</td>
<td>MIPS-3D</td>
<td>?</td>
<td>23 (fp)</td>
<td>FP 32x64b</td>
</tr>
<tr>
<td>AMD</td>
<td>E 3DNow!</td>
<td>99</td>
<td>24 (fp)</td>
<td>8x128 (new)</td>
</tr>
<tr>
<td>Intel</td>
<td>SSE-2</td>
<td>01</td>
<td>144 (int,fp)</td>
<td>8x128 (new)</td>
</tr>
</tbody>
</table>
Example of SIMD Operation (1)

**Sum of Partial Products**
Example of SIMD Operation (2)

**Pack (Int16→Int8)**

![Diagram of Pack operation](image)
Summary of SIMD Operations (1)

- **Integer arithmetic**
  - Addition and subtraction with saturation
  - Fixed-point rounding modes for multiply and shift
  - Sum of absolute differences
  - Multiply-add, multiplication with reduction
  - Min, max

- **Floating-point arithmetic**
  - Packed floating-point operations
  - Square root, reciprocal
  - Exception masks

- **Data communication**
  - Merge, insert, extract
  - Pack, unpack (width conversion)
  - Permute, shuffle
Summary of SIMD Operations (2)

- **Comparisons**
  - Integer and FP packed comparison
  - Compare absolute values
  - Element masks and bit vectors

- **Memory**
  - No new load-store instructions for short vector
    - No support for strides or indexing
  - Short vectors handled with 64b load and store instructions
  - Pack, unpack, shift, rotate, shuffle to handle alignment of narrow data-types within a wider one
  - Prefetch instructions for utilizing temporal locality
Programming with SIMD Extensions

- Optimized shared libraries
  - Written in assembly, distributed by vendor
  - Need well defined API for data format and use

- Language macros for variables and operations
  - C/C++ wrappers for short vector variables and function calls
  - Allows instruction scheduling and register allocation optimizations for specific processors
  - Lack of portability, non standard

- Compilers for SIMD extensions
  - No commercially available compiler so far
  - Problems
    - Language support for expressing fixed-point arithmetic and SIMD parallelism
    - Complicated model for loading/storing vectors
    - Frequent updates

- Assembly coding
SIMD Performance

Limitations

- Memory bandwidth
- Overhead of handling alignment and data width adjustments
A Closer Look at MMX/SSE

Higher speedup for kernels with narrow data where 128b SSE instructions can be used

Lower speedup for those with irregular or strided accesses
Vector Processors
Playstation 2 – Emotion Engine

The Emotion Engine

Control and Behavior/Emotion Synthesis
- FPU
- MIPS III CPU
- VU0

Geometry Processing
- VU1
- GIF

DMAC
IPU
RDRAM Interface
I/O Interface

to Graphics Synth
Playstation 2 - Emotion Engine (Vector Units)
Playstation 2 - Emotion Engine
Playstation 2 - Emotion Engine (Mips III)
Playstation 2 - Emotion Engine (Vector unit)
Vector Processors

- Initially developed for super-computing applications, but we will focus only on multimedia today.
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"
Properties of Vector Processors

- Single vector instruction implies lots of work (loop)
  - Fewer instruction fetches
- Each result independent of previous result
  - Compiler ensures no dependencies
  - Multiple operations can be executed in parallel
  - Simpler design, high clock rate
- Reduces branches and branch problems in pipelines
- Vector instructions access memory with known pattern
  - Effective prefetching
  - Amortize memory latency of over large number of elements
  - Can exploit a high bandwidth memory system
  - No (data) caches required!
Styles of Vector Architectures

Memory-memory vector processors
- All vector operations are memory to memory

Vector-register processors
- All vector operations between vector registers (except vector load and store)
- Vector equivalent of load-store architectures
- Includes all vector machines since late 1980s
- We assume vector-register for rest of the lecture
Components of a Vector Processor

- **Scalar CPU**: registers, datapaths, instruction fetch logic
- **Vector register**
  - Fixed length memory bank holding a single vector
  - Has at least 2 read and 1 write ports
  - Typically 8-32 vector registers, each holding 1 to 8 Kbits
  - Can be viewed as array of 64b, 32b, 16b, or 8b elements
- **Vector functional units (FUs)**
  - Fully pipelined, start new operation every clock
  - Typically 2 to 8 FUs: integer and FP
  - Multiple datapaths (pipelines) used for each unit to process multiple elements per cycle
- **Vector load-store units (LSUs)**
  - Fully pipelined unit to load or store a vector
  - Multiple elements fetched/stored per cycle
  - May have multiple LSUs
- **Cross-bar** to connect FUs, LSUs, registers
# Basic Vector Instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADD.VV</td>
<td>V1, V2, V3</td>
<td>V1 = V2 + V3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>VADD.SV</td>
<td>V1, R0, V2</td>
<td>V1 = R0 + V2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>VMUL.VV</td>
<td>V1, V2, V3</td>
<td>V1 = V2 x V3</td>
<td>vector x vector</td>
</tr>
<tr>
<td>VMUL.SV</td>
<td>V1, R0, V2</td>
<td>V1 = R0 x V2</td>
<td>scalar x vector</td>
</tr>
<tr>
<td>VLD</td>
<td>V1, R1</td>
<td>V1 = M[R1..R1+63]</td>
<td>load, stride=1</td>
</tr>
<tr>
<td>VLDS</td>
<td>V1, R1, R2</td>
<td>V1 = M[R1..R1+63*R2]</td>
<td>load, stride=R2</td>
</tr>
<tr>
<td>VLDX</td>
<td>V1, R1, V2</td>
<td>V1 = M[R1+V2i,i=0..63]</td>
<td>indexed(&quot;gather&quot;)</td>
</tr>
<tr>
<td>VST</td>
<td>V1, R1</td>
<td>M[R1..R1+63]=V1</td>
<td>store, stride=1</td>
</tr>
<tr>
<td>VSTS</td>
<td>V1, R1, R2</td>
<td>V1 = M[R1..R1+63*R2]</td>
<td>store, stride=R2</td>
</tr>
<tr>
<td>VSTX</td>
<td>V1, R1, V2</td>
<td>V1 = M[R1+V2i,i=0..63]</td>
<td>indexed(&quot;scatter&quot;)</td>
</tr>
</tbody>
</table>

+ all the regular scalar instructions (RISC style)…
Vector Memory Operations

Load/store operations move groups of data between registers and memory

Three types of addressing

- **Unit stride**
  - Fastest
- **Non-unit (constant) stride**
- **Indexed (gather-scatter)**
  - Vector equivalent of register indirect
  - Good for sparse arrays of data
  - Increases number of programs that vectorize

Support for various combinations of data widths in memory and registers

- \{.L,.W,.H,.B\} x \{64b, 32b, 16b, 8b\}
Vector Code Example

\[ Y[0:63] = Y[0:63] + a \times X[0:63] \]

<table>
<thead>
<tr>
<th>64 element SAXPY: scalar</th>
<th>64 element SAXPY: vector</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LD</strong> R0,a</td>
<td><strong>LD</strong> R0,a</td>
</tr>
<tr>
<td><strong>ADDI</strong> R4,Rx,#512</td>
<td><strong>ADDI</strong> Rx,Rx,#8</td>
</tr>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td><strong>LD</strong> R2, 0 (Rx)</td>
<td><strong>VLD</strong> V1,Rx</td>
</tr>
<tr>
<td><strong>MULTD</strong> R2,R0,R2</td>
<td><strong>VMUL.SV</strong> V2,R0,V1</td>
</tr>
<tr>
<td><strong>LD</strong> R4, 0 (Ry)</td>
<td><strong>VLD</strong> V3,Ry</td>
</tr>
<tr>
<td><strong>ADDD</strong> R4,R2,R4</td>
<td><strong>VADD.VV</strong> V4,V2,V3</td>
</tr>
<tr>
<td><strong>SD</strong> R4, 0 (Ry)</td>
<td><strong>VST</strong> Ry,V4</td>
</tr>
<tr>
<td><strong>ADDI</strong> Rx,Rx,#8</td>
<td></td>
</tr>
<tr>
<td><strong>ADDI</strong> Ry,Ry,#8</td>
<td></td>
</tr>
<tr>
<td><strong>SUB</strong> R20,R4,Rx</td>
<td></td>
</tr>
<tr>
<td><strong>BNZ</strong> R20,loop</td>
<td></td>
</tr>
</tbody>
</table>
Setting the Vector Length

- A vector register can hold some maximum number of elements for each data width (maximum vector length or MVL)
- What to do when the application vector length is not exactly MVL?
- Vector-length (VL) register controls the length of any vector operation, including a vector load or store
  - E.g. vadd.vv with VL=10 is
    for (I=0; I<10; I++) V1[I]=V2[I]+V3[I]
- VL can be anything from 0 to MVL
- How do you code an application where the vector length is not known until run-time?
Strip Mining

- Suppose application vector length > MVL
- **Strip mining**
  - Generation of a loop that handles MVL elements per iteration
  - A set operations on MVL elements is translated to a single vector instruction

**Example: vector saxpy of N elements**
- First loop handles \((N \mod MVL)\) elements, the rest handle MVL

```plaintext
VL = (N mod MVL);            // set VL = N mod MVL
for (I=0; I<VL; I++)          // 1st loop is a single set of
    Y[I]=A*X[I]+Y[I];      //   vector instructions
low = (N mod MVL);
VL = MVL;                     // set VL to MVL
for (I=low; I<N; I++)         // 2nd loop requires N/MVL
    Y[I]=A*X[I]+Y[I];      //   sets of vector instructions
```
Choosing the Data Type Width

Alternatives for selecting the width of elements in a vector register (64b, 32b, 16b, 8b)

Separate instructions for each width
- E.g. vadd64, vadd32, vadd16, vadd8
- Popular with SIMD extensions for GPPs
- Uses too many opcodes

Specify it in a control register
- Virtual-processor width (VPW)
- Updated only on width changes

NOTE
- MVL increases when width (VPW) gets narrower
- E.g. with 2Kbits for register, MVL is 32, 64, 128, 256 for 64-, 32-, 16-, 8-bit data respectively
- Always pick the narrowest VPW needed by the application
Other Features for Multimedia

- **Support for fixed-point arithmetic**
  - Saturation, rounding-modes etc

- **Permutation instructions of vector registers**
  - For reductions and FFTs
  - Not general permutations (too expensive)

- **Example: permutation for reductions**
  - Move 2\(^{nd}\) half a a vector register into another one
  - Repeatedly use with vadd to execute reduction
  - Vector length halved after each step

![Diagram](image)
Optimization 1: Chaining

Suppose:
- \texttt{vmul.vv V1,V2,V3}
- \texttt{vadd.vv V4,V1,V5}  \# RAW hazard

Chaining
- Vector register (V1) is not as a single entity but as a group of individual registers
- Pipeline forwarding can work on individual vector elements

Flexible chaining: allow vector to chain to any other active vector operation \(\Rightarrow\) more read/write ports
Optimization 2: Multi-lane Implementation

- Elements for vector registers interleaved across the lanes
- Each lane receives identical control
- Multiple element operations executed per cycle
- Modular, scalable design
- No need for inter-lane communication for most vector instructions
Chaining & Multi-lane Example

VL=16, 4 lanes, 2 FUs, 1 LSU, chaining -> 12 ops/cycle
Just one new instruction issued per cycle !!!!
Optimization 3: Conditional Execution

Suppose you want to vectorize this:

```c
for (I=0; I<N; I++)
    if (A[I]!= B[I]) A[I] -= B[I];
```

Solution: vector conditional execution

- Add **vector flag registers** with single-bit elements
- Use a **vector compare** to set the a flag register
- Use flag register as mask control for the vector sub
  - Addition executed only for vector elements with corresponding flag element set

Vector code

```assembly
vld V1, Ra
vld V2, Rb
vcmp.neq.vv F0, V1, V2  # vector compare
vsub.vv V3, V2, V1, F0  # conditional vsub
vst V3, Ra
```
Vector Architecture State

Virtual Processors ($mvl$)

General Purpose Registers
- $v_{r_0}$
- $v_{r_1}$
- $v_{r_{31}}$

Flag Registers (32)
- $v_{f_0}$
- $v_{f_1}$
- $v_{f_{31}}$

Scalar Registers
- $r_0$
- $r_1$
- $r_{31}$

$V_{pw}$ bits
64 bits
Two Ways to Vectorization

- **Inner loop vectorization**
  - Think of machine as, say, 32 vector registers each with 16 elements
  - 1 instruction updates 32 elements of 1 vector register
  - Good for vectorizing single-dimension arrays or regular kernels (e.g. saxpy)

- **Outer loop vectorization**
  - Think of machine as 16 “virtual processors” (VPs) each with 32 scalar registers! (multithreaded processor)
  - 1 instruction updates 1 scalar register in 16 VPs
  - Good for irregular kernels or kernels with loop-carried dependences in the inner loop

- **These are just two compiler perspectives**
  - The hardware is the same for both
Outer-loop Example (1)

// Matrix-matrix multiply:
//    sum a[i][t] * b[t][j] to get c[i][j]
for (i=1; i<n; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<n; t++)
        {
            sum += a[i][t] * b[t][j]; // loop-carried
        } // dependence
        c[i][j] = sum;
    }
}


Outer-loop Example (2)

// Outer-loop Matrix-matrix multiply:
// sum a[i][t] * b[t][j] to get c[i][j]
// 32 elements of the result calculated in parallel
// with each iteration of the j-loop (c[i][j:j+31])
for (i=1; i<n; i++) {
    for (j=1; j<n; j+=32) { // loop being vectorized
        sum[0:31] = 0;
        for (t=1; t<n; t++) {
            ascalar = a[i][t]; // scalar load
            bvector[0:31] = b[t][j:j+31]; // vector load
            prod[0:31] = b_vector[0:31]*ascalar; // vector mul
            sum[0:31] += prod[0:31]; // vector add
        }
        c[i][j:j+31] = sum[0:31]; // vector store
    }
}
Designing a Vector Processor

- Changes to scalar core
- How to pick the maximum vector length?
- How to pick the number of vector registers?
- Context switch overhead?
- Exception handling?
- Masking and flag instructions?
Changes to Scalar Processor

- Decode vector instructions
- Send scalar registers to vector unit (vector-scalar ops)
- Synchronization for results back from vector register, including exceptions
- Things that don’t run in vector don’t have high ILP, so can make scalar CPU simple
How to Pick Max. Vector Length?

♦ Vector length => Keep all VFUs busy:

\[
\text{Vector length} \geq \frac{\text{(\# lanes)} \times \text{(\# VFUs)}}{\text{\# Vector instr. issued/cycle}}
\]

♦ Notes:
  - Single instruction issue is always the simplest
  - Don’t forget you have to issue some scalar instructions as well
How to Pick Max Vector Length?

 Longer good because:

- Lower instruction bandwidth
- If know max length of app. is < max vector length, no strip mining overhead
- Tiled access to memory reduce scalar processor memory bandwidth needs
- Better spatial locality for memory access

 Longer not much help because:

- Diminishing returns on overhead savings as keep doubling number of elements
- Need natural app. vector length to match physical register length, or no help
- Area for multi-ported register file
How to Pick # of Vector Registers?

- More vector registers:
  - Reduces vector register “spills” (save/restore)
  - Aggressive scheduling of vector instructions: better compiling to take advantage of ILP

- Fewer
  - Fewer bits in instruction format (usually 3 fields)

- 32 vector registers are usually enough
Context Switch Overhead?

- The vector register file holds a huge amount of architectural state
  - To expensive to save and restore all on each context switch

- Extra dirty bit per processor
  - If vector registers not written, don’t need to save on context switch

- Extra valid bit per vector register, cleared on process start
  - Don’t need to restore on context switch until needed

- Extra tip:
  - Save/restore vector state only if the new context needs to issue vector instructions
Exception Handling: Arithmetic

- **Arithmetic traps are hard**
- **Precise interrupts => large performance loss**
  - Multimedia applications don’t care much about arithmetic traps anyway
- **Alternative model**
  - Store exception information in vector flag registers
  - A set flag bit indicates that the corresponding element operation caused an exception
  - Software inserts trap barrier instructions from SW to check the flag bits as needed
  - IEEE floating point requires 5 flag registers (5 types of traps)
Exception Handling: Page Faults

Page faults must be precise
- Instruction page faults not a problem
- Data page faults harder

Option 1: Save/restore internal vector unit state
- Freeze pipeline, (dump all vector state), fix fault, (restore state and) continue vector pipeline

Option 2: expand memory pipeline to check all addresses before send to memory
- Requires address and instruction buffers to avoid stalls during address checks
- On a page-fault on only needs to save state in those buffers
- Instructions that have cleared the buffer can be allowed to complete
Exception Handling: Interrupts

- Interrupts due to external sources
  - I/O, timers etc
- Handled by the scalar core
- Should the vector unit be interrupted?
  - Not immediately (no context switch)
  - Only if it causes an exception or the interrupt handler needs to execute a vector instruction
Vector Power Consumption

- Can trade-off parallelism for power
  - Power = C *Vdd² *f
  - If we double the lanes, peak performance doubles
  - Halving f restores peak performance but also allows halving of the Vdd
  - Power_{new} = (2C)*(Vdd/2)²*(f/2) = Power/4

- Simpler logic
  - Replicated control for all lanes
  - No multiple issue or dynamic execution logic

- Simpler to gate clocks
  - Each vector instruction explicitly describes all the resources it needs for a number of cycles
  - Conditional execution leads to further savings
Why Vectors for Multimedia?

- Natural match to parallelism in multimedia
  - Vector operations with VL the image or frame width
  - Easy to efficiently support vectors of narrow data types
- High performance at low cost
  - Multiple ops/cycle while issuing 1 instr/cycle
  - Multiple ops/cycle at low power consumption
  - Structured access pattern for registers and memory
- Scalable
  - Get higher performance by adding lanes without architecture modifications
- Compact code size
  - Describe N operations with 1 short instruction (v. VLIW)
- Predictable performance
  - No need for caches, no dynamic execution
- Mature, developed compiler technology
Comparison with SIMD

MORE SCALABLE
- Can use double the amount of HW (datapaths/registers) without modifying the architecture or increasing instruction issue bandwidth

SIMPLER HARDWARE
- A simple scalar core is enough
- Multiple operations per instruction

FULL SUPPORT FOR VECTOR LOADS AND STORES
- No overhead for alignment or data width mismatch

MATURE COMPILER TECHNOLOGY
- Although language problems are similar...

DISADVANTAGES
- Complexity of exception model
- Out of fashion...
**A Vector Media-Processor: VIRAM**

- **Technology: IBM SA-27E**
  - 0.18mm CMOS, 6 copper layers
  - 280 mm² die area
  - 158 mm² DRAM, 50 mm² logic
- **Transistor count: ~115M**
  - 14 Mbytes DRAM
- **Power supply & consumption**
  - 1.2V for logic, 1.8V for DRAM
  - 2W at 1.2V
- **Peak performance**
  - 1.6/3.2/6.4 Gops (64/32/16b ops)
  - 3.2/6.4/12.8 Gops (with madd)
  - 1.6 Gflops (single-precision)
- **Designed by 5 graduate students**

---

![Diagram of VIRAM architecture](image.png)
# Performance Comparison

<table>
<thead>
<tr>
<th>Task</th>
<th>VIRAM</th>
<th>MMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>iDCT</td>
<td>0.75</td>
<td>3.75 (5.0x)</td>
</tr>
<tr>
<td>Color Conversion</td>
<td>0.78</td>
<td>8.00 (10.2x)</td>
</tr>
<tr>
<td>Image Convolution</td>
<td>1.23</td>
<td>5.49 (4.5x)</td>
</tr>
<tr>
<td>QCIF (176x144)</td>
<td>7.1M</td>
<td>33M (4.6x)</td>
</tr>
<tr>
<td>CIF (352x288)</td>
<td>28M</td>
<td>140M (5.0x)</td>
</tr>
</tbody>
</table>

- QCIF and CIF numbers are in clock cycles per frame
- All other numbers are in clock cycles per pixel
- MMX results assume no first level cache misses
FFT (1)

FFT (Floating-point, 1024 points)

- VIRAM
- Pathfinder-2
- Wildstar
- TigerSHARC
- ADSP-21160
- TMS320C6701

Execution Time (usec)

<table>
<thead>
<tr>
<th>Device</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRAM</td>
<td>36</td>
</tr>
<tr>
<td>Pathfinder-2</td>
<td>16.8</td>
</tr>
<tr>
<td>Wildstar</td>
<td>25</td>
</tr>
<tr>
<td>TigerSHARC</td>
<td>69</td>
</tr>
<tr>
<td>ADSP-21160</td>
<td>92</td>
</tr>
<tr>
<td>TMS320C6701</td>
<td>124.3</td>
</tr>
</tbody>
</table>
FFT (2)
**SIMD Summary**

- **Narrow vector extensions for GPPs**
  - 64b or 128b registers as vectors of 32b, 16b, and 8b elements
- **Based on sub-word parallelism and partitioned datapaths**
- **Instructions**
  - Packed fixed- and floating-point, multiply-add, reductions
  - Pack, unpack, permutations
  - Limited memory support
- **2x to 4x performance improvement over base architecture**
  - Limited by memory bandwidth
- **Difficult to use (no compilers)**
Vector Summary

- Alternative model for explicitly expressing data parallelism
- If code is vectorizable, then simpler hardware, more power efficient, and better real-time model than out-of-order machines with SIMD support
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Will multimedia popularity revive vector architectures?