Streaming Vector Processing Unit:

Stream Processing Using SIMD on a General Purpose Processor

ELEC 525
Spring 2004

Michael Calhoun
Michael Chang
Manik Gadhiok
Marjan Karkooti

Project Hypothesis

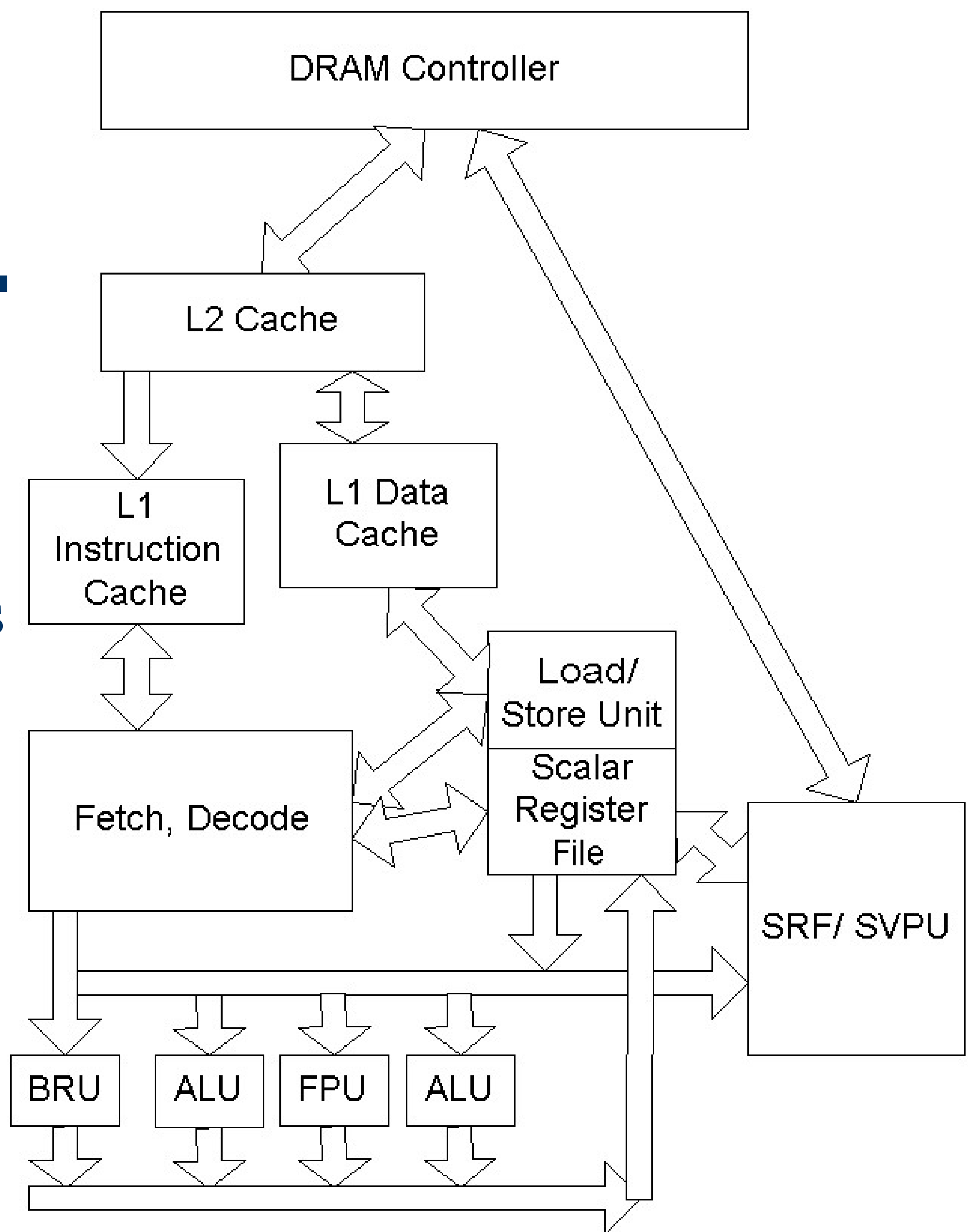
- Modern scalar processors inefficiently use fetch bandwidth when executing vectorizable code.
- By adding an SVPU we will
  - Increase the efficiency of the fetch system by reducing the number of instructions required to perform the desired computation
  - Simplify control logic for parallel execution
  - Stream register hierarchy improves memory efficiency by exploiting locality in data streams

Inefficiencies of Scalar Execution

- Scalar load op allows us to load any register from anywhere in memory
- However, most memory accesses and computation are sequential and predictable (loop bodies, array computations, media applications)
- We can combine these sequential memory accesses and computations into streams
- Extracting ILP becomes increasingly difficult as the width of processors scale; control logic, verification, and design are becoming exponentially complex

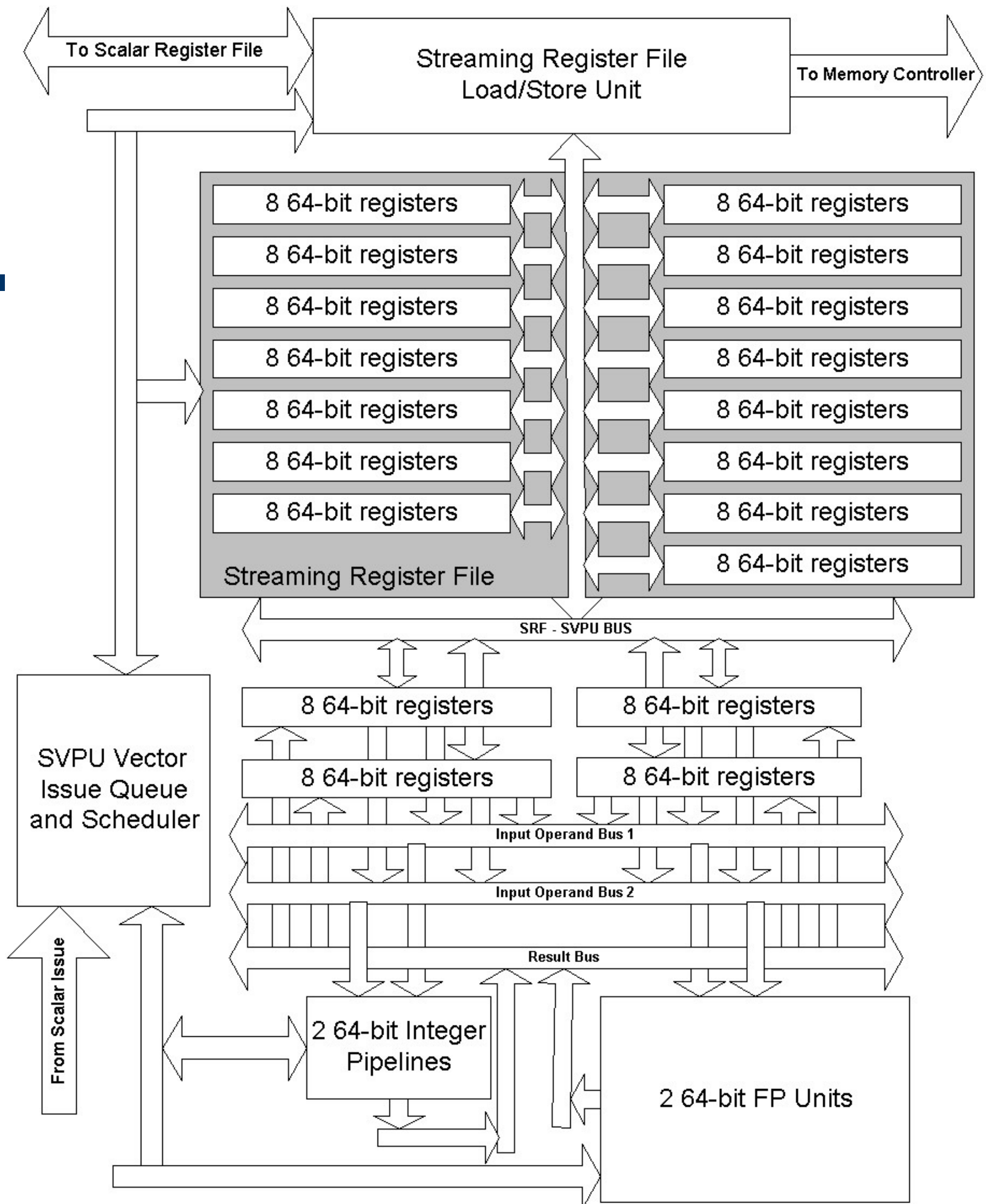
SVPU Architecture

- The SVPU is a special functional unit designed to execute vector instructions
- Receives stream instructions from issue logic and executes them
- Direct connection to DRAM controller, no cache access
- Has access to the scalar register file
- L2 Cache snoops accesses to the DRAM controller for correctness



SVPU

- 32 Stream registers, each 8 64-bit values
- 4 local registers
- 32 instruction Issue Queue and Scheduler
- 2 Pipelined Integer ALUs
- 2 Pipelined FP ALUs
- Load/Store Unit supports 8 outstanding operations



Vectorizable Code Finder

- Created Perl script to find vectorizable code segments.

C- CODE

```
@@@@ deflate.s @@@@
VECTORIZABLE CODE 3
===================
! 545 for (n = 0; n < HASH_SIZE; n++)
{ ! 547 head[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
```

Benchmark Results

GZIP	35 vec code segments found
MATEXP	11 vec code segments found
MPEG2DEC	17 vec code segments found

ASSEMBLY CODE

```
sethi %hi(32768),%l0
cmp %i4,%l0
sub %i4,%l0,%l0
mov %l0,%l2
mov %g0,%l2
sll %i5,1,%l0
sethi %hi(prev+65536),%l1
or %l1,%lo(prev+65536),%l1
sth %l2,[%l0+%l1]
add %i5,1,%i5
sethi %hi(32768),%l0
cmp %i5,%l0
blu .L317 nop
```

Vectorizable Profiling

- Profile of vectorizable code during dynamic execution.

```
Profile of vectorizable code for GZIP
================================
VecCode/Line#   Times Executed  Percent
vec unzip.c 154     40          ( 0.00)    = *
vec deflate.c 338   10          ( 0.00)    = *
vec deflate.c 545   5079040     (24.59)    = *************
vec deflate.c 550   5079040     (24.59)    = *************
vec inflate.c 309   6027        ( 0.03)    = *
vec inflate.c 325   183         ( 0.00)    = *
vec inflate.c 331   369         ( 0.00)    = *
vec spec.c 87       3           ( 0.00)    = *
vec spec.c 96       3072        ( 0.01)    = *
vec trees.c 350     256         ( 0.00)    = *
vec trees.c 365     256         ( 0.00)    = *
vec util.c 72       10485760    (50.77)    = **************************
```

Vectorizable Code Stats

	GZIP	MPEG2DEC	MAT8EXP
Vectorized Code Segs	3 of 35	2 of 17	9 of 11
Static Instructions Modified	45	48	135
Scalar Instructions Vectorized (Dynamic)	320.8M	2.1M	194.2M
Total Dynamic Instructions	1037.1M	14.1M	196M
Percent Vectorizable	30.9%	14.5%	99.1%

Example of Stream Instruction Replacement

ex.

```
LOOP:  add %g1,1,%g1;
       ld [MEM+%g1], %g2;
       add %g2, 10, %g2;
       st %g2, [MEM];
       ble %g1, 8,
```

<u>Replace w/ stream instructions (w/ 8 ALU's):</u>

```
       ldstream [MEM], %s1;
       addstream %s1, 10, %s1;
       ststream %s1, [MEM];
```

SimpleScalar Implementation

- Wrap each vectorizable code segment with annotated instructions
- Detect annotated instructions in the decode stage
- Remove the time taken for annotated block from total execution time

Stream Instruction Calculations

- Stream Size = 8 64-bit words
- Pipelined Memory Accesses
- Array sizes on benchmarks are greater than or equal to 32k elements

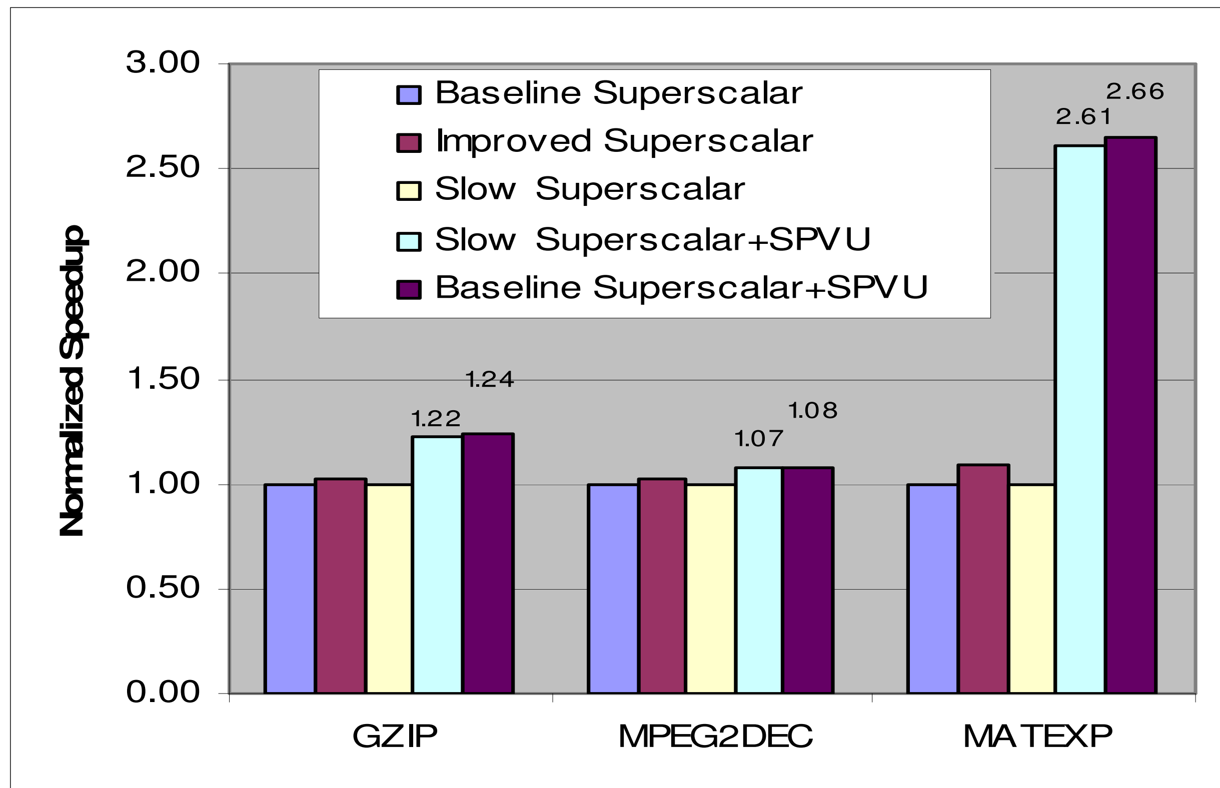
Stream Instructions	Latency (cycles)	Throughput (cycles/op)
Memory Instructions	100 or 200	4
SRF to LRF Instructions	2	2
INT ALU Operations	2	1
MUL Operations	4	1
FP Add/Sub/MUL	4	1
FP DIV	16	4

Architectural Comparison

- Comparison of 5 different architectural configurations, varying 4 parameters

	Int ALUs	FP ALUs	Memory Latency	Stream Processor
Baseline Superscalar	2	2	100 cycles	No
Improved Superscalar	4	4	100 cycles	No
Slow Superscalar	2	2	200 cycles	No
Baseline Superscalar with SVPU	2	2	100 cycles	Yes
Slow Superscalar with SVPU	2	2	200 cycles	Yes

Stream Unit Speedup Results



- Speedup is relative to baseline superscalar architecture

Conclusions

- Applications contain vectorizable code, but the percentage is variable
- SVPU shows significant speed-ups for certain applications
- Scalable solution that uses area and resources efficiently
- Significant compiler support required