Stream Vector Processing Unit: Stream Processing Using SIMD on a General Purpose Processor

ELEC525 – Spring 2004

Michael Calhoun
Michael Chang
Manik Gadhiok
Marjan Karkooti

Hypothesis

Modern scalar processors inefficiently use fetch bandwidth when executing vectorizable code. By augmenting a general-purpose processor with a Stream Vector Processing Unit (SVPU), we can use the fetch bandwidth much more efficiently and achieve speed-ups in performance on vectorizable code. We also believe that by using a hierarchy of register files, this architecture will use memory bandwidth more efficiently by exploiting locality in data streams.

Introduction

The world of general-purpose computing has experienced a transformation from text and 2-D processing to media processing. Multimedia workloads are becoming increasingly dominant in general-purpose computing [1], [2]. The performance of applications such as high-resolution games, video conferencing, signal processing, and image manipulation on a typical superscalar leaves a lot to be desired. Chipmakers like Intel, AMD and Sun have rolled out media SIMD-style ISA extensions (MMX [3], 3DNow! [4], VIS [5]) to meet the high computational demands of these applications and extract the data parallelism inherent in them. Fixed-function ASIC solutions also exist but the ISA extensions approach provides easier programmability, better performance, and easier upgrades from one generation to the next. Certain other applications, such as scientific code also exhibit data-level parallelism.

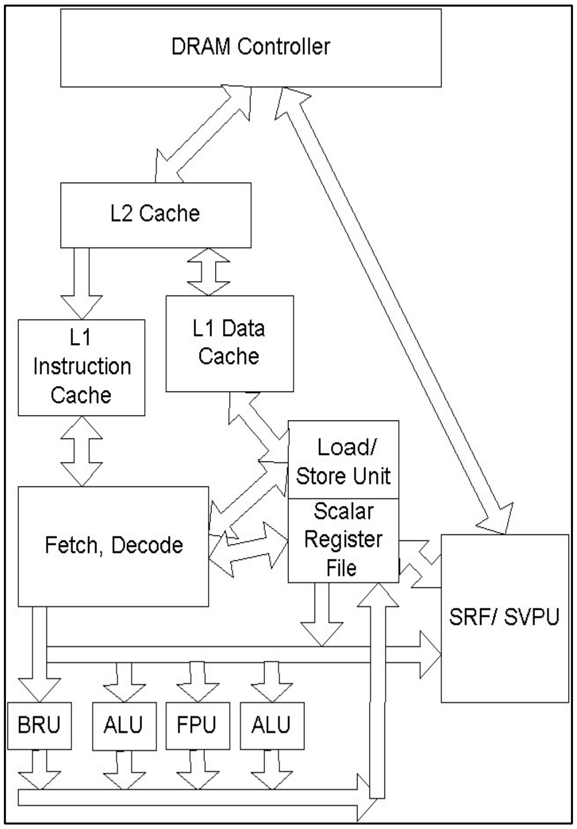
We propose to augment a general-purpose processor with a stream-vector processing unit (SVPU) to improve performance on vectorizable numerical and multimedia applications. By converting the data-parallel code segments of these programs into vector instructions, we believe that we can utilize the memory bandwidth much more efficiently. In the rest of the paper, we will also refer to these code segments (showing potential of conversion to vector instructions) as vectorizable code.

Ideas for the architecture for the SVPU are taken from the Imagine stream processor [6]. The SVPU is simpler, more scalable and complexity effective than existing architectures.

The proposed architecture gives us the benefits of a vector unit together with a wide-issue superscalar processor.

Architecture

Our project is the modification of a superscalar general-purpose processor. The basic core of the GPP is left intact, and we augment it with a new functional unit, the Stream Vector Processor Unit. The SVPU works by executing special, instruction-set defined instructions on streams of data loaded from a special compiler controlled cache.



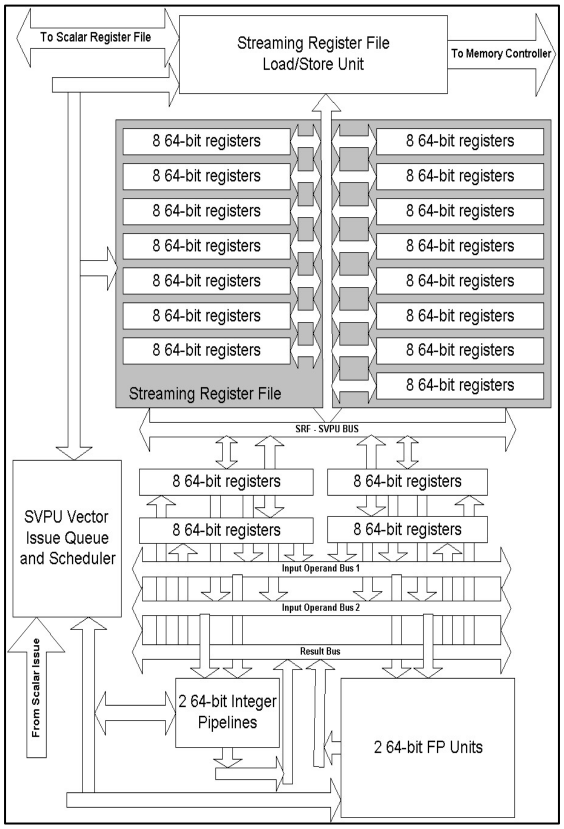


Figure 1: Block Diagram of Processor with an on-chip Stream Unit

Figure 2: Block Diagram of the Stream Unit (SRF/SVPU)

Core Architecture:

The basic GPP is a superscalar processor capable of fetching and issuing multiple instructions per clock cycle. Instructions are fetched, decoded, and then sent for execution down one of the different functional units (Integer ALU, FP ALU, BRU, LSU). Stream instructions are fetched and decoded exactly the same as scalar instructions, and then they are issued to the SVPU for processing (*Figure1*). Each instruction, including stream instructions, occupies a slot in the reorder buffer to ensure that the results commit in-order. Stream instructions stall in the issue logic when they depend on a value from the scalar core (ST.SSR, ST.LSR).

When a stream instruction finishes, it signals the ROB that it has completed and commits its value. However, since the result is not written on a global bus or stored anywhere, SVPU instructions do not natively support precise exceptions. If an exception occurs and a stream instruction has already committed a value out-of-order that would cause a problem, the ROB will have an entry for the instruction and can trap to the OS for appropriate handling.

The scalar register file can communicate with the SVPU, as certain instructions call for a read of GPR16 or a bulk register transfer from/to GPRs 8-15. The L1 instruction cache is read-only, and the L1 data cache is write-through so that the L2 cache contains all of the live copies and coherency information. The SVPU does not connect to the L1 or L2 caches of the superscalar core (only the memory controller), so the L2 cache must snoop the loads and stores to the memory controller and take appropriate coherency actions. The memory controller supports virtual channels and lazy-precharge to optimize for row reuse and memory access locality.

SVPU Architecture

The SVPU (*Figure 2*) is composed of the Stream Register File (SRF) and the Vector Processing Unit (VPU). The SRF is a compiler-controlled cache totaling 2KB in size composed of 32 stream registers ('streams'), each made of 8 64-bit type-agnostic data words. Attached to the SRF is the SVPU Load/Store Unit (LSU), which interfaces between the SRF and either the scalar register file or memory. The LSU supports 8 outstanding memory requests, and can issue requests in-order and receive serviced requests out of order. However, the LSU has to disambiguate memory addresses before allowing out of order commitment of results.

The VPU is the core of the SVPU and is responsible for the execution of arithmetic operations. The VPU has four local stream registers, each holding 8 64-bit values. Each of the local registers is organized into four banks of 64-bit pairs, and the local registers are the only ones that can directly feed the ALUs of the VPU. The VPU has two fully pipelined 64-bit integer arithmetic units and two pipelined floating-point units. The VPU performs operations on the granularity of streams, supporting three-operand format instructions that perform the same operation 8 or 16 times on data from one or more streams.

The SRF and VPU are fed instructions by the SVPU issue-queue/scheduler. Instructions enter the SVPU at the issue queue and are checked for dependencies using a basic scoreboard algorithm. When all dependencies are satisfied, the instruction can issue to the SRF LSU, VPU local register file, or the VPU core for execution. Instructions that use different resources can execute in parallel, i.e. ALU and memory operations. The SRF and LSU are connected by a 128-bit/cycle unidirectional bus, the SRF and VPU by a 256-bit/cycle unidirectional bus, and the scalar register file is connected to the LSU by a 64-bit unidirectional bus.

Instruction Set Extensions

We need to add 16 instructions to the basic ISA to implement the SVPU. We assume that we have a 32-bit instruction with 8 bits required for opcode (leaving 24 bits for instruction control data).

There are 7 integer ALU instructions (ADD, SUB, COMP, SHIFT, AND, OR, XOR).
Stream ALU Instruction Format:

[LR (dest) (2-bits)] [LR (source1) (2)] [LR (source2) (2)] [32-bit flag(1)] [use immediate (1)] [intra-stream op(1)] [immediate(15)]

The 64-bit ALUs can treat the 64-bit data word as two 32-bit values for these basic operations if this is set in a flag in the instruction. A 15-bit immediate is used instead of source2 if the use-immediate bit is set. The instruction performs the operation on each of the 8 data elements in the stream (source register 1) and accumulates the result in the element specified by the top three bits of the immediate. The operations proceed element wise starting at the accumulate element.

If the 32-bit flag is also set, the 8 64-bit registers in the stream are treated as 8 32-bit pairs, and the resulting 32-bit pair is written in the corresponding 64-bit register.
There are 5 floating point ALU instructions (MUL(integer), FP.ADD, FP.SUB, FP.MUL, FP.DIV). Floating Point Stream Instruction Format:

[LR(2) (dest)] [LR(2) (source1)] [LSR (2) (source2)] [32-bit flag (1)] [Intra-Stream Op (1)]

The FP units calculate on 64-bit (single precision) values by default, but if the 32-bit flag it set in the instruction, then the units treat pairs of registers in the stream as 32-bit single precision values for the FP ALUs, which natively support 64 and 32-bit FP math.
The instruction performs the operation on each of the 8 data elements in the local stream register (LR) and accumulates the result in the element specified by the top three bits of the immediate. The operations proceed element wise starting at the accumulate element.
If the 32-bit flag is also set, the 8 64-bit registers in the stream are treated as 8 32-bit pairs, and the resulting 32-bit pair is written in the corresponding 64-bit register.

There are 4 Memory and Register instructions, LSR (load stream register from memory), SSR (store stream register to memory), MSR (move stream/stream), and MSL (move stream/local). Memory and Register Instruction Format:

LSR – *[Destination Stream Reg (5 bits)] [Stride (6)] [Offset (13)]*

Using GPR16 as the base address + offset(13) , load 8 64-bit values into stream register (SR0 is not allowed), using stride specified from memory.

SSR – *[Source Stream Reg (5)] [Stride (6)] [Offset (13)]*

Using GPR16 as the base address + offset(13) , store 8 64-bit values from stream register (SR0 is not allowed), using stride specified into memory

MSR – *[Destination Stream Reg (5)] [source SR (5)] [Mask (8)]*

Move one stream register to another, using the mask to specify which registers to transfer (xFF is all of the registers). Stream register 0 (SR0) aliases to GPRs 8-15 and initiates a data transfer to the scalar register file if SR0 is the source or destination for this instruction. This transfer needs to be recognized by the scalar decode logic and use of the register file must be scheduled as a part of the scalar execution path.

MSL – *[To SRF (1)] [Local Register (2)] [Stream Register (5)] [Mask (8)]*

Moves a stream register to the local register for use in computation. SR0 is not allowed for the source/destination for the stream register. If the To SRF bit is set, the instruction copies the value from the specified local register to the specified stream register. The mask determines which of the registers within the stream register to copy.

Experimental Methodology

To evaluate our hypothesis accurately, we needed to easily find and quantify vectorizable code in different types of programs. We also needed to be able to accurately simulate execution time for the scalar code versus the stream version of that code. Having the capability to do both these things would allow us to go forward in our experiment to understand speedup improvements with an SVPU. A side benefit of this exercise would also give us some understanding of the ease or difficulty in recognizing vectorizable code for future work in compiler support.

We generated a Perl script, which could be run on C programs that were assembled using the "-S" option. The source code output from this compile option would include the C-code in comments and the actual assembly code.

Our script searches for the vectorizable C-code segments and prints out the corresponding assembly instructions. *Figure 3* shows an example of the script output.

C- CODE

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

Benchmark Results

GZIP	35 vec code segments found
MATEXP	9 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
```

Figure 3: Output of Vectorizable Code Finder

Once we found the vectorizable code segments, we wanted to concentrate on only converting the segments that were executed the most frequently. To achieve this, we used a combination of the gprof utility tool and our own profiling script. Using these tools, we could keep track of the number of times that a code segment was executed dynamically. This information allowed us to find out which segments we should convert into stream instructions. *Figure 4* shows an example of the output from our code profiling script.

```
◆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)  = **************************
```

Figure 4: Profile of how often each vectorizable code segment was run

The first and second column in *Figure 4*, describe which code segment and the number of times it was executed. The third column describes the percentage of times that code segment was run relative to the total number of code segments run.

From *Figure 4*, we can see that out of all the vectorizable code segments that we found, only 3 segments dominate the dynamic instruction count for GZIP. Thus, we could focus our efforts on converting only a few of the segments into stream instructions and still maximize the performance benefit. To attain the scalar execution time of a code segment we used Simplescalar to run the simulation.

<u>Simplescalar Simulations</u>

We performed all simulations using Simplescalar 3.0 [7]. We used the annotations capability of the toolset to simulate the SVPU enhanced superscalar processor. In the assembly code for the application, we marked the beginning and end of each vectorizable code segment with annotated instructions and modified the simplescalar source to detect these annotations in the decode stage. A cycle counter tracked the number of cycles we spent executing the code enclosed by the annotated instructions. Note that on our SVPU-enhanced processor, this vectorized code will not be executed in the superscalar pipeline but in the SVPU instead. The value of the cycle counter, therefore, is subtracted from the total execution time, and then by adding the cycles required to execute the vectorizable code on a stream-vector processor unit we get an estimate for the total cycles required to execute the same program on a SVPU-enhanced machine.

The next part of the experiment required converting the scalar code into stream instructions. We were able to do this by hand, since the number of code segments that actually needed to be converted into stream was quite small based on our profiling results. *Table 1* shows the latency and throughput values we used for our stream unit.

Stream Instructions	Latency (cycles)	Throughput (cycles/inst)
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

Table 1: Latency and throughput of stream instructions used in our simulation

After getting the stream execution time, we could compare the results to our scalar execution time, and calculate the speedup of the benchmarks when using an on-chip stream processing unit.

Experimental Analysis

Table 2 shows a list of different architecture configurations that we used to test our hypothesis. All of these are 4-issue superscalar architectures. There are four variables in our design: Number of Integer ALUs, Number of floating point ALUs, Memory access latency and whether or not SVPU exists.

	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

Table2: Different Architecture configurations for simulations

We chose these configuration parameters for several reasons. A stream processing unit would take up area that could possibly be used for additional functional units (integer and floating point ALU's). Thus, the comparison between an improved superscalar versus the default superscalar with the addition of a stream processing unit would make the performance comparisons more fair.

The different configurations for memory latency hope to show that the processor with the SVPU is more memory latency tolerant then the default super-scalar. This tolerance is due to the inherent efficiency of fetching and processing large data streams.

Based on the simulation results from simplescalar simulator, our stream processing unit shows a speed-up of 5%~200% over an improved superscalar architecture depending on the application. A summary of these results is shown in *Figure 5*.

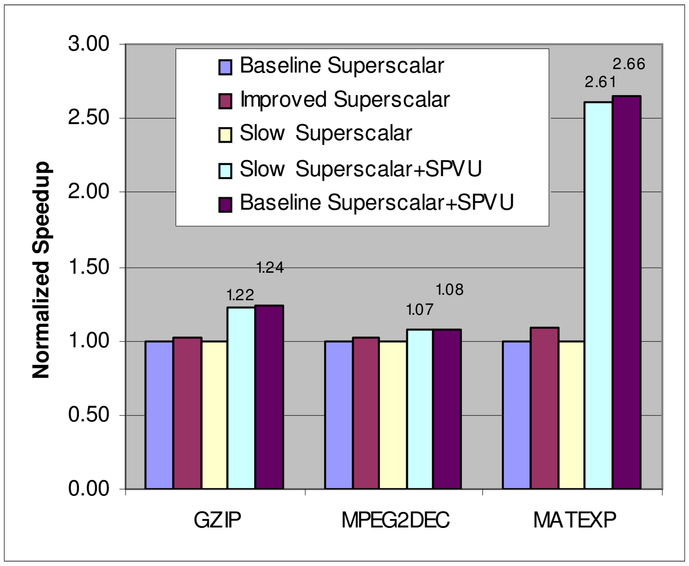


Figure 5: Speedup Results on 3 different benchmarks with various ALU and Memory Latency configurations

A breakdown of the different applications show that we can get speedups of 24% for GZIP, 8% for mpeg2dec and 266% for MATEXP micro-benchmark. This shows a significant advantage over the baseline architecture across various types of applications.

The MATEXP performs matrix exponentiation, which can be broken down into the highly data-parallel matrix multiplication and addition operations. As expected, it shows very good speedup. Since these operations occur very frequently in numerical, we expect an SVPU-enhanced superscalar to have large speedups for many numerical applications.

An interesting result was how MPEG2DEC, from mediabench [8], showed less speedup then we would have expected from a multimedia benchmark. We believe the reason is related to the lack of success in finding all the vectorizable code. The main computational kernel in MPEG2DEC is IDCT (inverse discrete cosine transform), which was written in a program style that made it difficult to find vectorizable code segments. In [9] the authors hand-optimized the IDCT code for MMX-extended ISA, and show a speed-up of approximately 2.3% compared to the Alpha ISA. This is the performance gain achieved after performing loop unrolling and software pipelining techniques and enhancing the MMX by providing independent register files and an increased number of local registers (32). Since they implement MMX instructions as library calls, the actual speedup can be expected to be higher than 2.3%. Another point worth noting is that the IDCT code runs on small block sizes (8). We can expect better speedup numbers as the block size increases.

Conversely, GZIP was written such that vectorizable code was relatively easy for our script to find, and achieved significant speedup results.

A conclusion drawn from this phenomenon is that the ease of exploiting vectorizable code can be variable depending on how the code is written. Given the free-form nature of writing code, this could give complications for future compilers unless specific rules are followed to allow explicit stream instruction translation.

Results on improved memory latency tolerance with a stream processing unit are somewhat inconclusive. Cache hit rates were very high for these benchmarks (greater than 99.2%), therefore, modeling memory latency tolerance accurately would probably require much larger data sets. The simulation of large data sets would have been desirable in understanding this difference; however, it was not feasible to do this given the timeframe of this project.

Given more time we would have liked to improve our vectorizable code finder script to find more subtle examples of SIMD parallelism. We would have also liked to model MMX performance versus our stream processing unit. Documented performance numbers on MMX show that it improved performance 13% on Intel's Media Benchmark, and 9.1% on 3DwinBench. Given our results, we believe that a stream vector processing unit would show similar or superior performance over MMX. Looking at the two architectures, the SVPU's compiler-controlled cache is decoupled from the scalar pipeline and can deliver high bandwidth to the vector unit. Also the vector instructions allow us to attain speed-ups even in the absence of sub-word parallelism. The MMX-extended ISA, though, has a feature where conditional branches can be converted into bit masks allowing parallel execution of the two possible control paths. While we expect MMX to get better performance than ours for those code segments (assuming similar branch prediction accuracies), given the same level of compiler support, the feature can be easily supported in our architecture.

Conclusions

The addition of a stream vector processing unit to a general purpose processor does show performance benefits with speedups ranging from 5% to 200% against an improved superscalar configuration.

The cost of adding a stream processing unit requires modifications to the ISA and requires significant compiler support. However, given the current limits of ILP, and the amount of data parallelism yet to be exploited, we feel that the addition of a stream processing unit is still a viable solution in improving single-threaded performance in future architectures.

Future work should include detailed comparisons between MMX versus stream processing performance. In addition, memory bandwidth simulations should be modeled to ensure what bandwidth requirements or solutions are necessary such that the stream processing unit and/or the general purpose processor are never starved.

References

[1] R. B. Lee and M. D. Smith. Media Processing: A New Design Target. In *IEEE MICRO*, pages 6–9, Aug 1996.

[2] K. Diefendorff and P. K. Dubey. How Multimedia Workloads Will Change Processor Design. In *IEEE*

Micro, pages 43–45, Sep 1997.

[3] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. In *IEEE Micro*, volume 16(4), pages 51–59, Aug 1996.

[4] S. Oberman et al. AMD 3DNow! Technology and the K6-2 Microprocessor. In *HOTCHIPS10*, 1998.

[5] M. Tremblay et al. VIS Speeds New Media Processing. In *IEEE Micro*, volume 16(4), pages 51–59, Aug 1996.

[6] Imagine: Media Processing with Streams, in *IEEE Micro*, 2001

[7] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin–Madison, June 1997 and documentation for all Simplescalar releases (through version 3.0).

[8] C. Lee, M. Potkonjak, and W. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," In *IEEE Micro*, vol. 30, no. 1, pp. 330-335, Dec. 1997

[9] Jesus Corbal, Roger Espasa and Mateo Valero. MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications, In *Proceedings of the ACM/IEEE SC99 Conference (SC'99)*