VGP393C – Week 9.1

- Agenda:
  - Quiz #3
  - Assignment #3 due
  - SIMD
    - Overview
    - Data types
    - SSE instructions
    - Compiler intrinsics
  - Start assignment #4

Vector Computers of the Past

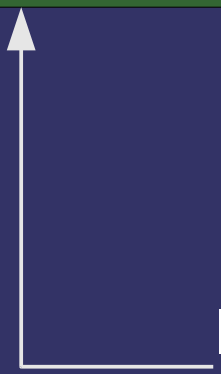
- Data would “stream” through special vector registers in a pipelined fashion

```
MOVE    VL, #64          ; Set vector length
VLOAD   VR0, A           ; Load first array
VLOAD   VR1, B           ; Load second array
VADD    VR2, VR1, VR0    ; Add all 64 elements
VSTORE  C, VR2           ; Store result
```

Vector Computers of the Past

- Data would “stream” through special vector registers in a pipelined fashion

```
MOVE    VL, #64          ; Set vector length
VLOAD   VR0, A           ; Load first array
VLOAD   VR1, B           ; Load second array
VADD    VR2, VR1, VR0    ; Add all 64 elements
VSTORE  C, VR2           ; Store result
```



Independent instructions can run on many “processors” at once

Vector Computers of the Past

- Vectorizing large sequential programs by hand is difficult and tedious
  - Compilers were created to do most of the work
  - Very effective for languages like FORTRAN

Vector Computers of the Past

- Vector architectures are difficult to combine with other modern techniques
  - Difficult to create vectored, out-of-order processors that fit in a desktop...or a cell phone

Remember the Cray-1?

Modern SIMD

- Compromise: add short, fixed-size vectors to existing architectures

Modern SIMD

- Explosion of interest in the mid- to late-90's:
  - UltraSPARC Visual Instruction Set (shipped 1995)
  - PA-RISC Multimedia Acceleration eXtensions (shipped 1995)
  - Pentium MultiMedia eXtensions (shipped 1997)
  - MIPS Digital Media eXtension (announced 1996)
  - Alpha Motion Video Instructions (shipped 1997)
  - PowerPC AltiVec (shipped 1998)
  - K6 3Dnow! (shipped 1998)
  - Pentium III SSE (shipped 1999)

Modern SIMD

- Explosion of interest in the mid- to late-90's:
  - UltraSPARC Visual Instruction Set (shipped 1995)
  - PA-RISC Multimedia Acceleration eXtensions (shipped 1995)
  - Pentium MultiMedia eXtensions (shipped 1997)
  - MIPS Digital Media eXtension (announced 1996)
  - Alpha Motion Video Instructions (shipped 1997)
  - PowerPC AltiVec (shi
  - K6 3Dnow! (shipped 1
  - Pentium III SSE (shipped 1999)

Promised to revolutionize 3D graphics on the PC... what happened?

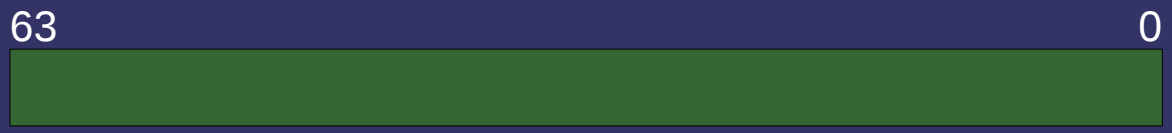
Modern SIMD

- Explosion of interest in the mid- to late-90's:
  - UltraSPARC Visual Instruction Set (shipped 1995)
  - PA-RISC Multimedia Acceleration eXtensions (shipped 1995)
  - Pentium MultiMedia eXtensions (shipped 1997)
  - MIPS Digital Media eXtension (announced 1996)
  - Alpha Motion Video Instructions (shipped 1997)
  - PowerPC AltiVec (shi
  - K6 3Dnow! (shipped 1
  - Pentium III SSE (shipped 1999)

3dfx released the first viable 3D accelerator for PCs

SIMD Registers

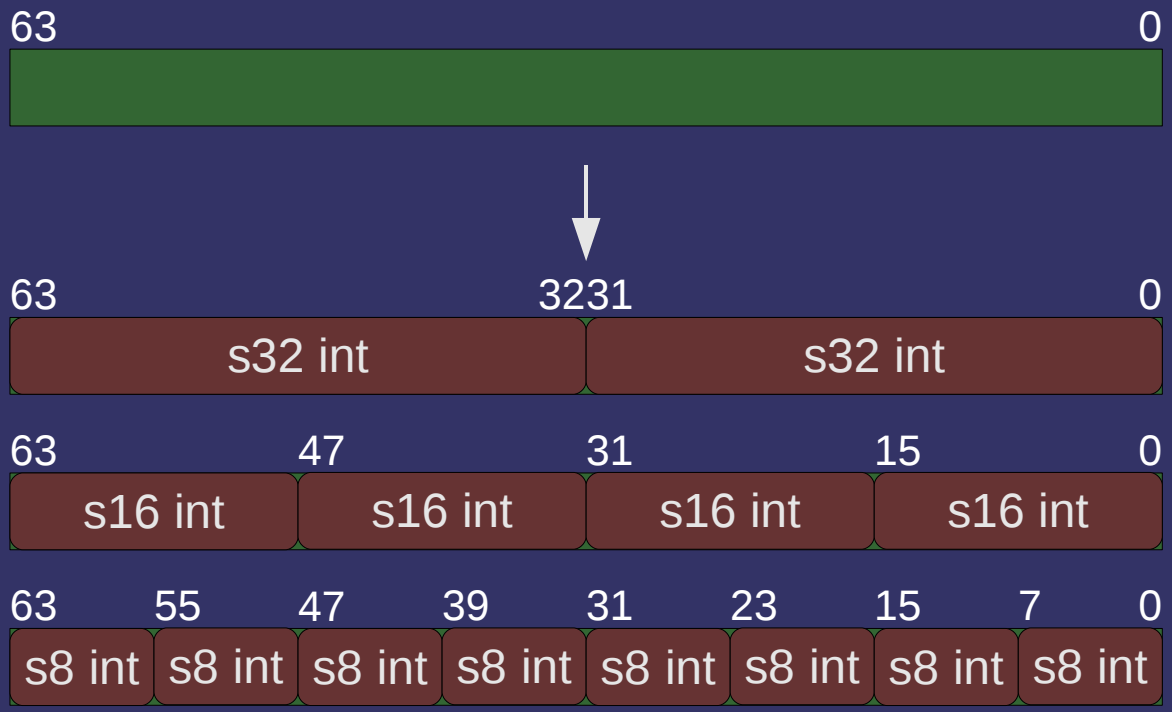
- All SIMD instruction sets utilize registers partitioned into multiple data items



64-bit register

SIMD Registers

- All SIMD instruction sets utilize registers partitioned into multiple data items



SIMD Registers

- All SIMD instruction sets utilize registers partitioned into multiple data items
  - Early implementations used 64-bit registers
  - Current implementations use 128-bit registers
  - Next-gen implementations will use 256-bit or 512-bit registers

SIMD Registers

- Early SIMD instruction sets reused floating-point registers as SIMD registers
  - Why?

SIMD Registers

- Early SIMD instruction sets reused floating-point registers as SIMD registers
  - Advantages:
    - Floating-point registers were already 64-bit while integer registers were typically only 32-bit
    - Operating systems already save and restore floating-point registers on context switch, so no OS changes were required
    - Re-use transistors already on the chip!
    - Re-use floating-point execution units
  - Disadvantages:
    - Can't mix floating-point and SIMD
    - Only 2-way SIMD for single precision float

Modern SIMD

- Explosion of interest in the mid- to late-90's:
  - UltraSPARC Visual Instruction Set (shipped 1995)
  - PA-RISC Multimedia Acceleration eXtensions (shipped 1995)
  - Pentium MultiMedia eXtensions (shipped 1997)
  - MIPS Digital Media eXtension (announced 1996)
  - Alpha Motion Video Instructions (shipped 1997)
  - PowerPC AltiVec (shipped 1998)
  - K6 3Dnow! (shipped 1998)
  - Pentium III SSE (shipped 1999)

Integer SIMD only!

Modern SIMD

- Explosion of interest in the mid- to late-90's:
  - UltraSPARC Visual Instruction Set (shipped 1995)
  - PA-RISC Multimedia Acceleration eXtensions (shipped 1995)
  - Pentium MultiMedia eXtensions (
  - MIPS Digital Media eXtension (
  - Alpha Motion Video Instructions (shipped 1997)
  - PowerPC AltiVec (shipped 1998)
  - K6 3Dnow! (shipped 1998)
  - Pentium III SSE (shipped 1999)

128-bit, dedicated SIMD registers

SIMD Data Types

- 128-bit SIMD instruction sets commonly have the following types:

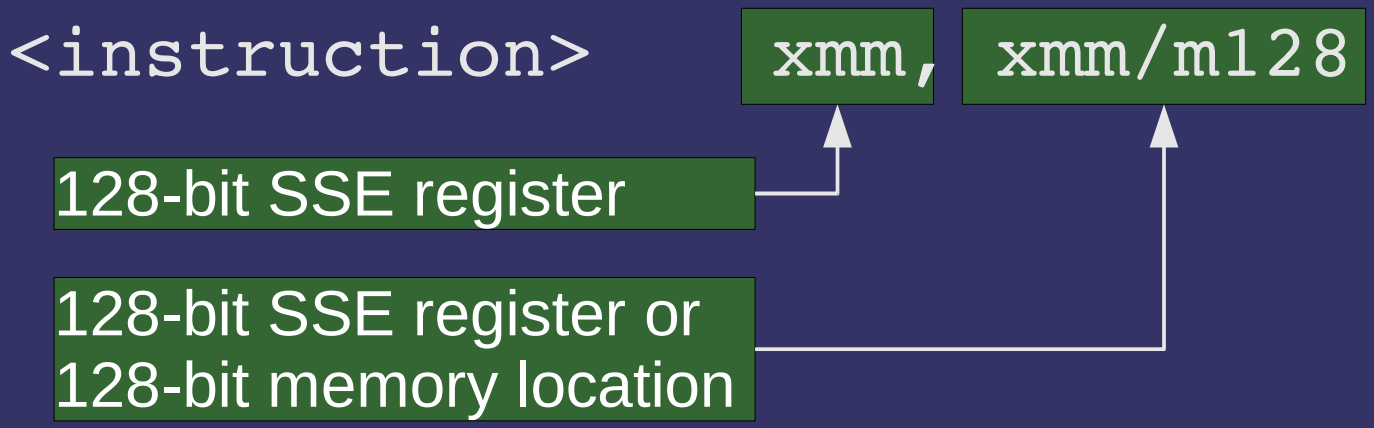
Type	Elements	Precision
signed bytes	16	8-bits
unsigned bytes	16	8-bits
signed short	8	16-bits
unsigned short	8	16-bits
signed long	4	32-bits
unsigned long	4	32-bits
signed long long	2	64-bits
unsigned long long	2	64-bits
float	4	32-bits
double	2	64-bits

SIMD Operations

- For the purpose of this class, only SSE and its successors will be considered
  - Other relevant SIMD architectures are similar

SIMD Operations

- Most SSE instructions use the following format:



Data Movement Instructions

Instruction	Suffix	Description
movdqa		Move double, 8-byte aligned
movdqu		Move double, unaligned
mova	ps, pd	Move floating-point aligned
movu	ps, pd	Move floating-point unaligned
movhl	ps	Move fp high to low
movlh	ps	Move fp low to high
movh	ps, pd	Move high packed fp
movl	ps, pd	Move low packed fp
mov	d, q, ss, sd	Move scalar
lddqu		Load 16-bytes, 8-byte aligned
mov<d/sh/sl>dup		Move and duplicate

Data Movement Instructions

- Extract a 32-bit value from a vector:

```
pextrw    r32, xmm, imm8
```

- Insert a 32-bit value into a vector:

```
pinsrw    xmm, r32, imm8
```

Selects a 32-bit portion of the 128-bit register

Data Movement Instructions

- Mask creation from vector:

```
pmovmskb    r32, xmm
movmskps    r32, xmm
movmskpd    r32, xmm
```

  - Creates a mask in `r32` of non-zero values in `xmm`
  - Mask is 8, 4, or 2 bits depending instruction variety

Arithmetic Instructions

Instruction	Suffix	Description
`padd`	`b, w, d, q`	Packed addition
`psub`	`b, w, d, q`	Packed subtraction
`padds`	`b, w`	Packed signed add w/saturati
`paddus`	`b, w`	Packed unsigned add w/satur
`psubs`	`b, w`	Packed signed sub w/saturati
`psubus`	`b, w`	Packed unsigned sub w/satur
`pmins`	`w`	Packed signed minimum
`pminu`	`b`	Packed unsigned minimum
`pmaxs`	`w`	Packed signed maximum
`pmaxu`	`b`	Packed unsigned maximum

Arithmetic Instructions

Instruction	Suffix	Description
`add`	`ss, ps, sd, pd`	Addition
`sub`	`ss, ps, sd, pd`	Subtraction
`div`	`ss, ps, sd, pd`	Division
`mul`	`ss, ps, sd, pd`	Multiplication
`min`	`ss, ps, sd, pd`	Minimum
`max`	`ss, ps, sd, pd`	Maximum
`sqrt`	`ss, ps, sd, pd`	Square root
`rcp`	`ss, ps`	Approx reciprocal
`rsqrt`	`ss, ps`	Approx reciprocal square root

Arithmetic Instructions

Instruction	Suffix	Description
`pavg`	`b, w`	Average w/rounding
`pmulh`	`w`	Signed mult high
`pmulhu`	`w`	Unsigned mult high
`pmull`	`w`	Mult low
`psad`	`bw`	Unsigned sum of absolute diff
`pmadd`	`wd`	Signed multiply and add
`addsub`	`ps, pd`	fp add / subtract
`hadd`	`ps, pd`	fp horizontal add
`hsub`	`ps, pd`	fp horizontal subtract

Arithmetic Instructions

- Horizontal add and subtract:

```
haddpd      xmm0, xmm1
```

xmm0	a1	a2
xmm1	b1	b2
xmm0	b1+ b2	a1 + a2

- Add / subtract:

```
addsubpd    xmm0, xmm1
```

a1	a2
b1	b2
b1 + a1	b2 - a2

Logical Instructions

Instruction	Suffix	Description
`pand`		Logical and
`pandn`		Logical and-not
`por`		Logical or
`pxor`		Logical exclusive-or
`and`	`ps, pd`	Logical and
`andn`	`ps, pd`	Logical and-not
`or`	`ps, pd`	Logical or
`xor`	`ps, pd`	Logical exclusive-or

Comparison Instructions

Instruction	Suffix	Description
`pcmp<cc>`	`b, w, d`	Integer compare
`cmp<cc>`	`ss, ps, sd, pd`	Floating-point compare

- Integer compare condition code, `<cc>`, can be equal (`eq`) or greater-than (`gt`)
- Floating-point compare condition code, `<cc>`, can be:

`eq`	Equal	`lt`	Less-than	`le`	Less-or-equal	`unord`	Unordered
`ne`	Not equal	`nlt / ge`	Not less-than	`nle / gt`	Not less-or-equal	`ord`	Ordered

Conversion Instructions

Instruction	Suffix	Description
`packss`	`wb, dw`	Pack signed w/saturate
`packus`	`wb`	Pack unsigned w/saturate
`cvt<s2d>`		Conversion
`cvtt<s2d>`		Conversion w/truncate

Conversion Instructions

- Floating-point / integer conversion modes, `<s2d>`, can be one of:
  - `dq2pd` – Two signed longs to double-precision FP
  - `pd2dq` – Two double-precision FP to signed long
  - `dq2ps` – Four signed long to single-precision FP
  - `ps2dq` – Four single-precision FP to signed long

Conversion Instructions

- Single-precision / double-precision conversion modes, `<s2d>`, can be one of:
  - `pd2ps` – Two double-precision FP to single-precision
  - `ps2pd` – Two single-precision FP to double-precision
  - `sd2ss` – One double-precision FP to single-precision
  - `ss2sd` – One single-precision FP to double-precision

Shift Instructions

Instruction	Suffix	Description
`psll`	`w, d, q, dq`	Shift left logical
`psra`	`w, d`	Shift right arithmetic
`psrl`	`w, d, q, dq`	Shift right logical

- Shift instructions take the form:

```
<instruction>    xmm, imm8
```

Shuffle Instructions

Instruction	Suffix	Description
`pshuf`	`w, d`	Shuffle
`pshufh`	`w, d`	Shuffle high
`pshufl`	`w`	Shuffle low
`shuf`	`ps, pd`	Shuffle

Shuffle Instructions

- Shuffle instructions take the form:

```
<instruction>    xmm, xmm/m128, imm8
```

  - The `imm8` value is a list of 2-bit values that select which source fields go to destination fields
    - Bits 0 and 1 of `imm8` for `pshufhw` select which 16-bit values from the high 64-bits of the source go to each field of the destination

```
pshufhw    xmm0, xmm1, 00011011b
```

xmm0	a7	a6	a5	a4	a3	a2	a1	a0
xmm1	b7	b6	b5	b4	b3	b2	b1	b0
imm8	0	1	10	11				
xmm0	b4	b5	b6	b7	a3	a2	a1	a0

Unpack Instructions

Instruction	Suffix	Description
`punpckh`	`bw, wd, dq, qdq`	Unpack high
`punpckl`	`bw, wd, dq, qdq`	Unpack low
`unpckh`	`ps, pd`	Unpack high
`unpckl`	`ps, pd`	Unpack low

- Unpack instructions pull values from two inputs and store either the high or low alternating halves in the destination

```
punpckhwd    xmm0, xmm1
```

xmm0	a7	a6	a5	a4	a3	a2	a1	a0
xmm1	b7	b6	b5	b4	b3	b2	b1	b0
xmm0	b7	a7	b6	a6	b5	a5	b4	a4

Break

Compiler Intrinsics

- Special types added for vectors:
  - `__mm128` – Four single-precision FP values
  - `__mm128i` – Four 32-bit integer values
  - `__mm128d` – Two double-precision FP values

Compiler Intrinsics

- Most instructions have built-in functions of the form:

```
__mm_<instruction>_<suffix>(...)
```

  - `addps` is:

```
__m128 __mm_add_ps(__m128 a, __m128 b);
```

  - As usual, MSDN has the full details

References

Corrina G. Lee. “ Short Vector Extensions in Commercial Microprocessors.” 21 Nov 1998. Accessed 9 Sept 2008 http://www.eecg.toronto.edu/~corinna/vector/svx/.

Alex Fr. “Introduction to SSE Programming.” 10 July 2008. Access 10 Sept 2008 http://www.codeproject.com/KB/recipes/sseintro.aspx.

Vectorizing Code

- Initial temptation is to use SIMD registers like GLSL `vec4`
  - This is often called *array-of-structures* (AoS) organization

Vectorizing Code

- AoS suffers from a number of problems on most SIMD architectures
  - Swizzle / shuffle operations can be expensive
  - Intra-register operations may be lacking
    - Dot-product instruction was only added in SSE4.1...first available in **2007** on the Penryn / Yorkdale cores (Intel) and Barcelona core (AMD)

Vectorizing Code

- Data would “stream” through special vector registers in a pipelined fashion

```
MOVE     VL, #64        ; Set vector length
VLOAD    VR0, A         ; Load first array
VLOAD    VR1, B         ; Load second array
VADD     VR2, VR1, VR0  ; Add all 64 elements
VSTORE   C, VR2         ; Store result
```

Vector length is fixed at 4 for single-precision FP

Vectorizing Code

- Data would “stream” through special vector registers in a pipelined fashion

```
    MOVE    ECX, #64 / 4; Set vector length
L:  MOVPS   XMM0, A       ; Load first array
    ADDPS   XMM0, B       ; Add second array
    MOVPS   C, XMM0       ; Store result
    ADD     A, #16        ; Increment pointers
    ADD     B, #16
    ADD     C, #16
    DEC     ECX
    JNZ     L
```

Vectorizing Code

- Instead of acting like the SIMD is a `vec4`, think of it as four grouped scalars
  - Think of the data in *transpose*

xmm0 x0 y0 z0 w0
xmm1 x1 y1 z1 w1
xmm2 x2 y2 z2 w2
xmm3 x3 y3 z3 w3

Vectorizing Code

- Instead of acting like the SIMD is a `vec4`, think of it as four grouped scalars
  - Think of the data in *transpose*

Vectorizing Code

- Transposed data like this is often called *structure-of-arrays* (SoA)
  - This works best when the source data is stored in this format

```
struct {
    float x;
    float y;
    float z;
    float w;
} foo[256];
```

Vectorizing Code

- Transposed data like this is often called *structure-of-arrays* (SoA)
  - This works best when the source data is stored in this format

```
struct {
    float x;
    float y;
    float z;
    float w;
} foo[256];
```

Becomes

```
struct {
    float x[256];
    float y[256];
    float z[256];
    float w[256];
} foo;
```

Data Alignment

- Most SSE instructions *require* that data by 16-byte (quad-doubleword) aligned
  - Force alignment of data using compiler extensions

```
__declspec(align(16)) float data[384];
```

    - Different compilers and different platforms do this differently
  - `__m128` and related types have the alignment magic
  - Use a special allocator to get properly aligned dynamic allocations

Next week...

- And by next week I mean *Friday 9/12*
- Intel Threaded Building Block
- Review for final
- Work on assignment #4

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.

10-September-2008