Multiprocessing

- Flynn’s Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- SISD: Single I Stream, Single D Stream
  - A uniprocessor
- SIMD: Single I, Multiple D Streams
  - Each “processor” works on its own data
  - But all execute the same instrs in lockstep
  - E.g. a vector processor or MMX, CUDA
Flynn’s Taxonomy

- MISD: Multiple I, Single D Stream
  - Not used much
  - Stream processors are closest to MISD
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - This is your typical off-the-shelf multiprocessor (made using a bunch of “normal” processors)
  - Includes multi-core processors
SIMD
SIMD Model

- Texas C62xx, IA32 (SSE), AMD K6, CUDA, Xbox..
- Early SIMD machines: e.g.) CM-2 (large distributed system)
  - Lack of vector register files and efficient transposition support in the memory system.
  - Lack of irregular indexed memory accesses
- Modern SIMD machines:
  - SIMD engine is in the same die
SIMD Execution Model

Source 1

<table>
<thead>
<tr>
<th></th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
</table>

Source 2

<table>
<thead>
<tr>
<th></th>
<th>Y3</th>
<th>Y2</th>
<th>Y1</th>
<th>Y0</th>
</tr>
</thead>
</table>

OP

OP

OP

OP

Destination

<table>
<thead>
<tr>
<th></th>
<th>X3 OP Y3</th>
<th>X2 OP Y2</th>
<th>X1 OP Y1</th>
<th>X0 OP Y0</th>
</tr>
</thead>
</table>

for (ii = 0; ii < 4; ii++)

x[ii] = y[ii] + z[ii];

SIMD_ADD(X, Y, Z)
MMX/SSE/3D Now!

- MMX:
  - Virtual registers. Use FP registers
  - Handles only integers. Designed for 2D graphics
- SSE: Introduce new registers
  (XMM0..XMM7, XMM15)
- 3DNow! (AMD)
**MMX Instruction Set**

- Arithmetic
  - Addition, subtraction, multiplication, multiply and Add
- Comparison
  - Compare for equal, greater than
- Conversion
  - Pack
- Unpack
- Logical
  - And, And Not, Or, Exclusive OR
- Data Transfer
  - Register to register, load from memory
SSE

- New data type
  - 128-bit packed single-precision floating-point data type

- Packed/Scalar single-precision floating-point instruction
- 64-bit SIMD integer instruction
- State management instructions
- Cacheability control, prefetch, and memory ordering instructions
SSE Packed and Scalar Floating-Point Instructions

Packed single-precision floating-point operation

Scalar single-precision floating-point operation
Shuffle and Unpack Instructions

Source 1
X3  X2  X1  X0

Source 2
Y3  Y2  Y1  Y0

Destination
Y3...Y0  Y3...Y0  X1  X0  OP  Y0

Scalar single-precision floating-point operation
PREFETCH\textsubscript{h} Instruction

- PREFETCH\textsubscript{h}: permits programs to load data into the processor at a suggested cache level.

CPU

Temporal/Non temporal

1\textsuperscript{st} level cache

2\textsuperscript{nd} level cache
SSE2/SSE3/SSE4

- Add new data types
- Add more complex SIMD instructions
- Additional vector registers
- Additional cacheability-control and instruction-ordering instructions.
SIMD Programming:

- Using compiler: Vectorising using compiler
- Assembler intrinsic:
  - E.g.) __m128_mm_add_ps (__m128 a, __m128b)
  - Translated one for one into equivalent assembler instructions
  - Pros: No need to worry about C optimization
  - Cons:
    - Not portable between processors
    - Requires the programmer to know the underlying machine architecture
    - Longer programming time
Loop unrolling

for (i = 1; i < 12; i++) x[i] = j[i]+1;

for (i = 1; i < 12; i=i+4)
{
    x[i] = j[i]+1;
    x[i+1] = j[i+1]+1;
    x[i+2] = j[i+2]+1;
    x[i+3] = j[i+3]+1;
}

SSE ADD
Next Generation Micro Architecture
Intel® Core™2 Duo Processor

Shared L2 = 4MB

<table>
<thead>
<tr>
<th></th>
<th>CPU-0</th>
<th>CPU-0</th>
<th>CPU-1</th>
<th>CPU-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1D=32KB</td>
<td>L1I=32KB</td>
<td>L1D=32KB</td>
<td>L1I=32KB</td>
</tr>
<tr>
<td></td>
<td>LO/L1 DTB PMH</td>
<td>LO/L1 DTB PMH</td>
<td>LO/L1 DTB PMH</td>
<td>LO/L1 DTB PMH</td>
</tr>
<tr>
<td></td>
<td>CPU-0 Core</td>
<td>CPU-0 Core</td>
<td>CPU-1 Core</td>
<td>CPU-1 Core</td>
</tr>
</tbody>
</table>
SIMD PROGRAMMING IN GRAPHICS

(compilation, architectural support, and evaluation of SIMD graphics pipeline programs on a general-purpose CPU)
• Vertex shader: operates on a vertex.
  – Transform step: converts element coordinates from one frame of reference to another e.g., between world space and eye space
  – Lighting....
• Pixel shader(fragment shader): rasterization phase. Processes a pixel
  – Texturing, filtering
Example: Vertex Shader

\[
\begin{align*}
\text{dp4} \& \text{oPos.x, v0, c[0]} \\
\text{dp4} \& \text{oPos.y, v0, c[1]} \\
\text{dp4} \& \text{oPos.z, v0, c[2]} \\
\text{dp4} \& \text{oPos.w, v0, c[3]}
\end{align*}
\]

Apply a constant color to each vertex

\[\text{dp4}: \text{a four-component dot-product}\]

\[
\text{mov oD0, c[4]}
\]

Mask: selects components of a vertex that are not affected by the operation.
Swizzle: reorders and/or replicates components of the vector operands to the instruction.
Vertex Shader Program Using SSE

- Shader instruction: `dp4 oPos, v0, v1`
  - A dot product of v0 and v1 and stores the result in each of the four words of oPos

SSE2

Input: %xmm0, %xmm1, %xmm2 output
- `movaps %xmm5, %xmm0`
- `mulps %xmm5, %xmm1`
- `pshufd %xmm6, %xmm5, $14`
- `addps %xmm6, %xmm5`
- `pshufd %xmm5, %xmm6, $1`
- `addps %xmm5, %xmm6`
- `pshufd %xmm2, %xmm5, $0`
Vertex Shader Program Using SSE-II

- shader instruction: `mul r0.xz, v0.xyz, v1.w`
  multiplies \([v0.x, v0.y, v0.z, v0.z]\), \([v1.w, v1.w, v1.w, v1.w]\)
  stores the x and z components of the result into r0.

  \[= \text{mul} \ r0.x\_z\_\_ \ v0.xyzz, v1.wwww\]

SSE:

- `pshufd %xmm5, %xmm0, $164`
- `pshufd %xmm6, %xmm1, $255`
- `mulps %xmm6, %xmm5`
- `andps %xmm6, _Mask+80`
- `andps %xmm2, _Mask+160`
- `orps %xmm2, %xmm6`
SOA & AOS

- Array of structures (AOS)
  - \{x1,y1, z1,w1\}, \{x2,y2, z2,w2\}, \{x3,y3, z3,w3\}
  - \{x4,y4, z4,w4\} ....
  - Intuitive but less efficient
  - What if we want to perform only x axis?

- Structure of array (SOA)
  - \{x1,x2,x3,x4\}, ...,{y1,y2,y3,y4}, ...{z1,z2,z3,z4},
  - ... \{w1,w2,w3,w4\}...