| **course** | CS594: Scientific Computing for Engineers |
| **topic** | Short Vector SIMD Programming |
| **speaker** | Jakub Kurzak, PhD |
| **institution** | Electrical Engineering and Computer Science University of Tennessee Knoxville, Tennessee |
Topics

- heads up on the homework
- brief overview of the Cell chip
- overview of the short vector SIMD cores
- SIMD programming principles
- step-by-step SIMD example
- more on the homework
Homework

\[ C = C - A \times B^T \]

```c
void spe_matmul_trans(float *A, float *B, float *C)
{
    int m, n, k;

    for (m = 0; m < 64; m++)
        for (n = 0; n < 64; n++)
            for (k = 0; k < 64; k++)
                C[m*64+n] -= A[m*64+k] * B[n*64+k];
}
```

**objective:** vectorize & optimize
Introducing the CELL

Starting in 2001 design a microprocessor, which when ready in 2005, will outperform by an order of magnitude other top of the line microprocessors.

- 5 years
- 400 people
- $400 million
- Sony/Toshiba/IBM alliance known as STI
- STI Design Center - Austin, Texas - March 2001
Introducing the CELL

- multi-core
- in-order execution
- shallow pipeline
- SIMD
- scratchpad memory

- 3.2 GHz
- 90 nm OSI
- 234 M transistors
  - 165 M – Xbox 360
  - 731 M – quad Intel Core i7 (2008)
  - 2,000 M – quad Intel Tukwila (2009 ?)

- 204.8 GFLOPS single precision
- 14.6 GFLOPS double precision
- 204.8 GB/s internal bandwidth
- 25.6 GB/s memory bandwidth
## CELL Performance

### Single Precision Floating Point
- PowerPC core: 25.6 GFLOPS
- SIMD core: 25.6 GFLOPS
- All SIMD cores: 204.8 GFLOPS

### Double Precision Floating Point
- PowerPC core: 6.4 GFLOPS
- SIMD core: 1.8 GFLOPS
- All SIMD cores: 14.6 GFLOPS

### Memory Bandwidth
- Main memory: 25.6 GB/s
- SIMD core: 25.6 GB/s
- Interconnect: 204.8 GB/s

---

**one 2005 CBE**

- at 3.2 GHz

**two 2009 Xeons**

- Quad-core at 3.2 GHz

*Single precision peak using SSE on Xeons*
CELL Hardware

- Mercury PCI accelerator cards
- IBM server blades
- Sony Playstation 3
- Embedded systems
In June 2008 The Roadrunner supercomputer at Los Alamos National Laboratory crossed the performance of one PetaFLOPS ($10^{15}$ floating point operations per second) using 6,480 AMD Opteron dual-core processors and 12,960 IBM PowerXCell 8i processors.
CELL Overview

- **PPE** – Power Processing Element
- **SPE** – Synergistic Processing Element
- **SPU** – Synergistic Processing Unit
- **LS** – Local Store
- **MFC** – Memory Flow Controller
- **EIB** – Element Interconnection Bus
- **MIC** – Memory Interface Controller
Synergistic Processing Element

- SIMD architecture
- Two in-order (dual issue) pipelines
- Large register file (128 128-bit registers)
- 256 KB of scratchpad memory (Local Store)
- Memory Flow Controller to DMA code and data from system memory
Local Store Design

Local Store
- fully pipelined
- single ported
- **256 KB SRAM**
- (18-bit addressing)
- untranslated
- unguarded
- non-corehent
- **16 B loads and stores**
- **fixed 6-cycle latency**
- no exceptions
PlayStation 3

- 1 CELL
- 6 SPEs
  - one physically destroyed
  - one reserved by the hypervisor
- 256 MB RAM
- GigE
PlayStation 3 Clusters

- Linux (Fedora Core 5-7)
- OpenMPI
- GigE
- 256 MB RAM

Google “SCOP3”
some say

Real programmers code in assembly !!!
Assembly

actually

Relax, we will do business in C
Short Vector SIMD

- 2 doubles – including FMA – – not fully pipelined – – 4 FLOP / 7 cycles
- 4 floats – including FMA – – fully pipelined – – 8 FLOP / cycle
- 4 integers
- 8 short integers
- 16 chars
SPE User Registers

in the SPU ABI:

- **0** – dedicated – link register
- **1** – dedicated – stack pointer
- **2** – volatile – environment pointer (for languages that need one)
- **3-79** – volatile – function arguments, return values, and general usage
- **80-127** – non-volatile – used for local variables
  must be preserved across function calls
Preferred Slot

- addresses
- lengths of shifts and rotations
- .....
## SIMD Data Types

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>Sixteen 8-bit unsigned chars</td>
</tr>
<tr>
<td>vector signed char</td>
<td>Sixteen 8-bit signed chars</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>Eight 16-bit unsigned halfwords</td>
</tr>
<tr>
<td>vector signed short</td>
<td>Eight 16-bit signed halfwords</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>Four 32-bit unsigned words</td>
</tr>
<tr>
<td>vector signed int</td>
<td>Four 32-bit signed words</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>Two 64-bit unsigned doublewords</td>
</tr>
<tr>
<td>vector signed long long</td>
<td>Two 64-bit signed doublewords</td>
</tr>
<tr>
<td>vector float</td>
<td>Four 32-bit single-precision floats</td>
</tr>
<tr>
<td>vector double</td>
<td>Two 64-bit double precision floats</td>
</tr>
<tr>
<td>qword</td>
<td>quadword (16-byte)</td>
</tr>
</tbody>
</table>
SPU Intrinsics

- **specific intrinsics** – `si_.....` – **deprecated in favor of assembly**
  - intrinsics that have a one-to-one mapping with a single assembly-language instruction

- **generic intrinsics** – `spu_.....`
  - intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters

- **composite intrinsics** – `spu_.....`
  - convenience intrinsics constructed from a sequence of specific or generic intrinsics

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spu_mfcdma32(ls, ea, size, tagid, cmd)</code></td>
<td>Initiate DMA to or from 32-bit effective address</td>
</tr>
<tr>
<td><code>spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)</code></td>
<td>Initiate DMA to or from 64-bit effective address</td>
</tr>
<tr>
<td><code>spu_mfcstat(type)</code></td>
<td>Read MFC tag status</td>
</tr>
</tbody>
</table>
si_ Intrinsics

- one-to-one mapping with assembly
- SPU assembly instruction prefixed with si_
- also available as generic intrinsics except for:
  - generate controls for sub-quadword insertions
  - constant formation
  - NOPs
  - load and store
  - control
- accompanied by casting si_ intrinsics – no effect on the data

### Example:

<table>
<thead>
<tr>
<th>Constant Formation Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = si_ii(imm)</td>
</tr>
<tr>
<td>d = si_iia(imm)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Operation Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>si_inop()</td>
</tr>
<tr>
<td>si_nop()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Load and Store Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = si_lq(a, imm)</td>
</tr>
<tr>
<td>d = si_lqa(imm)</td>
</tr>
</tbody>
</table>
# spu_Intrinsics

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant Formation Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_splats(a)</code></td>
<td>Replicate scalar a into all elements of vector d</td>
</tr>
<tr>
<td><strong>Conversion Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_convtf(a, scale)</code></td>
<td>Convert integer vector to float vector</td>
</tr>
<tr>
<td><strong>Arithmetic Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_add(a, b)</code></td>
<td>Vector add</td>
</tr>
<tr>
<td><strong>Byte Operation Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_avg(a, b)</code></td>
<td>Vector average</td>
</tr>
<tr>
<td><strong>Compare, Branch, and Halt Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_cmpeq(a, b)</code></td>
<td>Vector compare equal</td>
</tr>
<tr>
<td><strong>Bit and Mask Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_shuffle(a, b, pattern)</code></td>
<td>Shuffle bytes of a vector</td>
</tr>
<tr>
<td><strong>Logical Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_and(a, b)</code></td>
<td>Vector bit-wise AND</td>
</tr>
<tr>
<td><strong>Rotate Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_rl(a, count)</code></td>
<td>Element-wise bit rotate left</td>
</tr>
<tr>
<td><strong>Shift Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_sl(a, count)</code></td>
<td>Element-wise bit shift left</td>
</tr>
<tr>
<td><strong>Control Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>(void) spu_idisable()</code></td>
<td>Disable interrupts</td>
</tr>
<tr>
<td><strong>Scalar Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_extract(a, element)</code></td>
<td>Extract vector element from vector</td>
</tr>
<tr>
<td><strong>Channel Control Intrinsic</strong></td>
<td></td>
</tr>
<tr>
<td><code>d = spu_readch(channel)</code></td>
<td>Read word channel</td>
</tr>
</tbody>
</table>
Pick data structures with SIMD'zation in mind

```c
union {
    struct _coordinate {
        float x, y, z, w;
    } coordinate;
    vector float vertex;
} array[];
```

```c
struct {
    float x[];
    float y[];
    float z[];
    float w[];
} vertices;
```
Vector Element Manipulation

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>e</th>
<th>i</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>f</td>
<td>j</td>
<td>n</td>
</tr>
<tr>
<td>c</td>
<td>g</td>
<td>k</td>
<td>o</td>
</tr>
<tr>
<td>d</td>
<td>h</td>
<td>l</td>
<td>p</td>
</tr>
</tbody>
</table>

? -> ?
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>n</th>
<th>k</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>b</td>
<td>o</td>
<td>l</td>
</tr>
<tr>
<td>i</td>
<td>f</td>
<td>c</td>
<td>p</td>
</tr>
<tr>
<td>m</td>
<td>j</td>
<td>g</td>
<td>d</td>
</tr>
</tbody>
</table>

? -> ?
```
Byte Shuffle

- splats
- lookup table
- transposition
- endian reversal
- transform from AOS to SOA

Shuffle is an odd pipeline instruction
Byte Shuffle - 4 × 4 Transposition

vector unsigned char shufflehi = (vector unsigned char)
{0x00, 0x01, 0x02, 0x03, 0x10, 0x11, 0x12, 0x13,
  0x04, 0x05, 0x06, 0x07, 0x14, 0x15, 0x16, 0x17};

vector unsigned char shufflelo = (vector unsigned char)
{0x08, 0x09, 0x0A, 0x0B, 0x18, 0x19, 0x1A, 0x1B,
  0x0C, 0x0D, 0x0E, 0x0F, 0x1C, 0x1D, 0x1E, 0x1F};

aibj = spu_shuffle(abcd, ijkl, shufflehi);
ckdl = spu_shuffle(abcd, ijkl, shufflelo);
emfn = spu_shuffle(efgh, mnop, shufflehi);
gohp = spu_shuffle(efgh, mnop, shufflelo);

aeim = spu_shuffle(aibj, emfn, shufflehi);
bfjn = spu_shuffle(aibj, emfn, shufflelo);
cgko = spu_shuffle(ckdl, gohp, shufflehi);
dhlp = spu_shuffle(ckdl, gohp, shufflelo);

Cannot be done with a bit select. Has to be done with a shuffle.
Bit Select

vc = spu_sel(va, vb, vt);
• where vt == 0, put va in vc
• where vt == 1, put vb in vc

- bit masking
- branch avoidance
- data reorganization
- pointer manipulation

Bit select is an even pipeline instruction
Can be done with either bit select or shuffle.

```c
vector unsigned char sell0101 = (vector unsigned char)
{0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
  0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF};

vector unsigned char sell0011 = (vector unsigned char)
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

abkl = spu_sel(abcd, ijkl, sell0011);    ankh = spu_sel(abkl, mngh, sell0101);
efop = spu_sel(efgh, nmop, sell0011);    ebol = spu_sel(efop, abkl, sell0101);
ijcd = spu_sel(ijkl, abcd, sell0011);    ifcp = spu_sel(ijcd, efop, sell0101);
mngh = spu_sel(nmop, efgh, sell0011);    mjgd = spu_sel(mngh, ijcd, sell0101);
```
Shifts and Rotations

- only shift / rotate left
- register or immediate

- quadword (odd pipeline)
  - by bits – max 7 positions
  - by bytes – max 32 positions – shift by more than 15 bits zeros the quadword

- element (even pipeline)
  - word (32 bits)
    - by bits – 64 positions – shift by more than 31 bits zeros the word
  - halfword (16 bits)
    - by bits – 32 positions – shift by more than 15 bits zeros the halfword
### Dual Issue

- hazard detection and stall
- dual-issue rules
  - independent instructions
  - even pipeline – even work address
  - odd pipeline – odd word address

<table>
<thead>
<tr>
<th>Pipe 0 Instructions</th>
<th>Stall (clocks)</th>
<th>Latency (clocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision floating-point operations</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Integer multiplies, convert between float/integer, interpolate</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Immediate loads, logical ops, integer add/subtract, signed extend, count leading zeros, select bits, carry/borrow generate</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Double precision floating-point operations</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Element rotates and shifts</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Byte operations - count ones, abs difference, average, sum</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipe 1 Instructions</th>
<th>Stall (clocks)</th>
<th>Latency (clocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shuffle bytes, quadword rotates and shifts</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Gather, mask, generate insertion control</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Estimate</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Loads/stores, branch hints</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Branches</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Channel operations, move to/from SPRs</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
SPU Optimization Example

c = c - A^T \times b
void spu_code_tile(float *A, float *B, float *C) {
    int m, n;

    for (n = 0; n < 32; n++)
        for (m = 0; m < 32; m++)
            C[m] -= A[n*32+m] * B[n];
}

standard C
- good news: will compile and run
- bad news: no performance whatsoever
SPU Optimization Example

0D 01
1D 0123
0D 12
1D 1
0D --45
1D 456789
0 56
0 67
0D 78
1D 789012
0D 89
1D 890123
0D 90
1D 901234
1 0123
1 --3456
1 4567
1 5678
0 67
0 --901234
1 -----5678
1 --901234
0D 01
1D 0123
0 12
0d 23
0d 34
1d --5678
1
ori $14,$4,0
shlqbyi $15,$3,0
il $13,0
lnop
ori $8,$15,0
hbraq .L11,.L3
il $6,0
il $9,32
ori $3,$6,$5
lqx $12,$6,$5
a $2,$13,$14
lqd $4,0(38)
a $9,9,-1
lqx $11,$13,$14
cwx $10,$6,$5
rotqby $3,$12,$3
rotqby $4,$4,$8
rotqby $2,$11,$2
ai $8,$8,4
fnms $3,$4,$2,$3
shufb $7,$3,$12,$10
stqx $7,$6,$5
ai $6,$6,4
brnz $9,.L3
ai $13,$13,4
ai $15,$15,128
ceqi $2,$13,128
brz $2,.L2
bi $lr

0.24 GFLOPS
< 1% peak!!!
c = c − A^T × b
void spu_code_tile(float *A, float *B, float *C) {
    vector float *Ap = (vector float*)A;
    vector float *Cp = (vector float*)C;
    int m, n;

    for (n = 0; n < 32; n++)
    {
        vector float b_splat = spu_splats(B[n]);

        for (m = 0; m < 8; m++)
        {
            Cp[m] = spu_nmsub(Ap[n*8+m], b_splat, Cp[m]);
        }
    }
}"
SPU Optimization Example

```
0D 01
1D 0123
0D 12
1D 1234
0D 23
1D 2
0D  -45
1D 456789
0D  56
1D 567890
0d  67
1d  -----1234
0d  2
1d  -- -5678

0D  67
1D 678901
0d  78
1d  -890123
0d  -----456789
1d  -----012345

0D  12
1D 1234
0D  23
0d  34
0d  45
1d  -- 6789
1d  7890
```

```
ori       $10,$3,0
shlqbyi   $12,$4,0
il        $11,0
fsmbi     $9,0
ila       $13,66051
lnop

shufb     $8,$2,$2,$13
fnms      $3,$3,$8,$4
stqx      $3,$6,$5
ai        $7,$7,-1
lqx        $3,$9,$2

a         $2,$10,$6
lqx        $4,$6,$5
ai        $7,$7,$-1
lqx        $3,$9,$2

1.08 GFLOPS
4.5 × faster
~4% peak
```
void spu_code_tile(float *A, float *B, float *C) {
    vector float *Ap = (vector float*)A;
    vector float *Cp = (vector float*)C;
    vector float c0, c1, c2, c3, c4, c5, c6, c7, c8;
    int n;

    c0 = Cp[ 0];   c4 = Cp[ 4];
    c1 = Cp[ 1];   c5 = Cp[ 5];
    c2 = Cp[ 2];   c6 = Cp[ 6];
    c3 = Cp[ 3];   c7 = Cp[ 7];

    for (n = 0; n < 32; n++)
    {
        vector float b_splat = spu_splats(B[n]);

        c0 = spu_nmsub(Ap[n*8+ 0], b_splat, c0);
        c1 = spu_nmsub(Ap[n*8+ 1], b_splat, c1);
        c2 = spu_nmsub(Ap[n*8+ 2], b_splat, c2);
        c3 = spu_nmsub(Ap[n*8+ 3], b_splat, c3);
        c4 = spu_nmsub(Ap[n*8+ 4], b_splat, c4);
        c5 = spu_nmsub(Ap[n*8+ 5], b_splat, c5);
        c6 = spu_nmsub(Ap[n*8+ 6], b_splat, c6);
        c7 = spu_nmsub(Ap[n*8+ 7], b_splat, c7);
    }

    Cp[ 0] = c0;   Cp[ 4] = c4;
    Cp[ 1] = c1;   Cp[ 5] = c5;
}
SPU Optimization Example

7.64 GFLOPS

32 × faster
~30% peak
SPU Optimization Example

further optimizations

➢ loop pipelining / register double buffering
➢ compute even iteration / load store odd iteration

➢ loop tiling for improved locality
➢ loop linearization
➢ replace nested loops with a single loop + pointer arithmetic
SPU Optimization Example

```c
spu_splats(B[0]);

for (n = 0; n < 32; n+=2) {
    d_splat = spu_splats(B[n+1]);
    c0  = spu_nmsub(a0, b_splat, c0);
    c1  = spu_nmsub(a1, b_splat, c1);
    c2  = spu_nmsub(a2, b_splat, c2);
    c3  = spu_nmsub(a3, b_splat, c3);
    c4  = spu_nmsub(a4, b_splat, c4);
    c5  = spu_nmsub(a5, b_splat, c5);
    c6  = spu_nmsub(a6, b_splat, c6);
    c7  = spu_nmsub(a7, b_splat, c7);

    b_splat = spu_splats(B[n+2]);
    c0  = spu_nmsub(e0, d_splat, c0);
    c1  = spu_nmsub(e1, d_splat, c1);
    c2  = spu_nmsub(e2, d_splat, c2);
    c3  = spu_nmsub(e3, d_splat, c3);
    c4  = spu_nmsub(e4, d_splat, c4);
    c5  = spu_nmsub(e5, d_splat, c5);
    c6  = spu_nmsub(e6, d_splat, c6);
    c7  = spu_nmsub(e7, d_splat, c7);
}

Cp[0] = c0;    Cp[1] = c1;    Cp[2] = c2;    Cp[3] = c3;
```
### SPU Optimization Example

<table>
<thead>
<tr>
<th>Source</th>
<th>Instruction</th>
<th>Size 1</th>
<th>Size 2</th>
<th>Size 3</th>
<th>Size 4</th>
<th>Size 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L2:</td>
<td>$3,$30,$35</td>
<td>$2,$28,$26</td>
<td>$5,$3,4</td>
<td>$21,$3,8</td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td>0</td>
<td>L2:</td>
<td>$3,$30,$35</td>
<td>$2,$28,$26</td>
<td>$5,$3,4</td>
<td>$21,$3,8</td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lqd</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>fnms</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>lnop</td>
<td>$20,$22,$4,$20</td>
<td>$3,$2,$5</td>
<td>$6,$34,$4,$6</td>
<td><img src="image" alt="" /></td>
<td></td>
</tr>
</tbody>
</table>

**12.34 GFLOPS**

~51 × faster

~48% peak
Homework

\[ C = C - A \times B^T \]

```c
void spe_matmul_trans(float *A, float *B, float *C) {
    int m, n, k;

    for (m = 0; m < 64; m++)
        for (n = 0; n < 64; n++)
            for (k = 0; k < 64; k++)
                C[m*64+n] -= A[m*64+k] * B[n*64+k];
}
```
Homework

- matrix-matrix multiplication \( C = C - A \times B^T \)
- A, B and C - 64 x 64, single precision

- Provide your implementation of the `spe_matmul_trans()` function
- Original implementation achieves the performance of 0.26 Gflop/s
- Your goal is to speed it up by at least a factor of 10 and achieve 2.6 Gflop/s
- There will be no partial credit for performance below 2.0 Gflop/s

- The theoretical peak is 25.6 Gflop/s
- The best known implementation runs at 24 Gflop/s

- You are only asked to run at
  - less than 11 % of the peak
  - more than 9 times slower then the best code

- Achieving 2.75 Gflop/s is doable with
  - 3 intrinsic functions
  - 65 lines of code
Homework

- accounts on a Playstation 3: ig.eecs.utk.edu
- optimize the function `spe_matmul_trans()` in `spe_matmul.c`
- email the function body to kurzak@eecs.utk.edu
- deadline is March 25th, noon (strict)
- contest: try to do it with the smallest possible amount of code (smallest `spe_matmul.o`)
- prize: ICL swag