Programming the Cell BE

Part 1

Aim of this lecture

- The aim of this lecture is to...
  - ...get familiar with the Cell Broadband Engine,
  - ...understand SIMD concepts and programming,
  - ...learn common loop optimizations,
  - ...learn how to use the QS21 Cell blades at HPC2N,
  - ...introduce Assignment 4.
- The aim of this lecture is to...
  - ...introduce the MFC and communication,
  - ...learn how to use a static performance analysis tool,
  - ...introduce the SoA / AoS storage formats,
  - ...discuss programming models.

Glossary

- Multicore
  - Several independent cores (cf. CPU) physically placed on a single silicon chip and each one capable of running one or more threads.
- ILP
  - Instruction Level Parallelism.
- SPE / SPU
  - Synergistic Processing Element / Unit, a small 128-bit SIMD processing unit (SPU) with 256KB local store and MFC (SPE).
- PPE / PPU
  - Power PC Element / Unit, a large Power PC based core (PPU) with caches (PPE).
- LS
  - Local Store, 256KB software managed scratchpad memory.
- MFC
  - Memory Flow Controller, the software managed DMA engine in an SPE.
- EIB
  - Element Interconnect Bus, connects the SPEs, PPE, and main memory.
- FMA
  - Fused Multiply-Add, a single instruction performing a multiplication and addition.
- DMA
  - Direct Memory Access, a method in which the CPU is bypassed when accessing memory from other units and devices.
- EA
  - Effective Address, an address in an application's memory space.

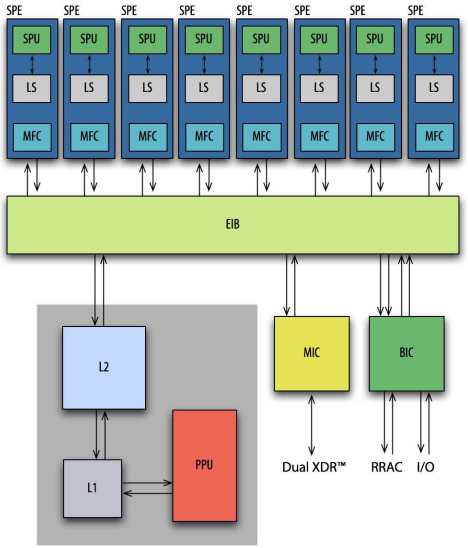
Multicore Processors

- Multicore processors are the standard today
- Three quick reasons why:
- The Power Wall
  - Power dissipation from chips is a limiting issue
    - High energy costs
    - High cooling costs
- The Frequency Wall
  - Deeper pipelines to ramp up frequency
    - Relies on ILP (limited resource in applications)
  - Higher frequency increases power dissipation
- The Memory Wall
  - Latency to DRAM is approaching 1000 cycles on large SMPs.
  - Performance is limited by the activity of moving data around.

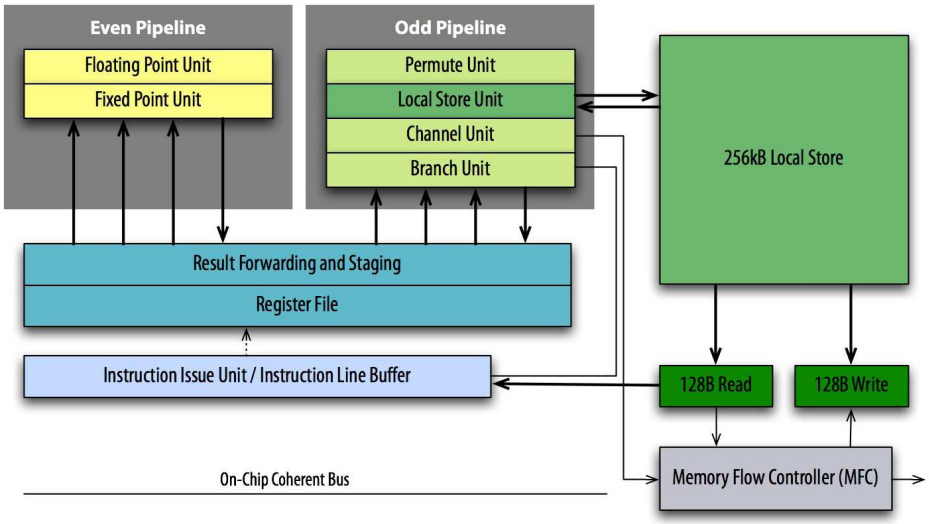
Multicore Processors

- How multicores address these issues:
- The Power Wall
  - Signals need to travel shorter distances in a small core
  - Frequency can be reduced and still increase performance
- The Frequency Wall
  - Shallower pipelines, decreased frequency, less power dissipation.
  - More cores, more work in parallel, better performance with less frequency.
- The Memory Wall
  - On-chip latency and bandwidth is order of magnitude better than off-chip (SMPs).
  - *However, all cores share the same memory interface...*

The Cell BE Architecture



The Synergistic Processing Element (SPE)



The SPE in numbers

- 256 KB local store
- 128 128-bit general purpose registers
- 128-bit SIMD architecture
- 128-bit / cycle LS bandwidth (25.6 GB/s)
- 25.6 GB/s main memory (DMA)
- EIB bandwidth 204.8 GB/s
- Instruction latency: 2-7 cycles (shallow pipelines)
- 2 pipelines

The Local Store

- The local store is likely the most exotic feature of the Cell
- Can be thought of as a software managed cache.
- All the complexity of caching and coherence is now the responsibility of the programmer.
  - Bad: can be difficult to learn and get right.
  - Good: exposes bad parallelization, memory traffic explicit.

- **What is wrong with regular caches?**

- Caches are reactive:
  - A cache miss when the data is needed, pipeline stalls
  - Cache replacement policies → unpredictable performance
  - Hardware prefetching (typically managed by compiler)
- Local store is managed:
  - Data movement is explicit
  - Programmer notices bad implementation details
  - Programmer can use expert knowledge to overlap memory traffic

What is the peak performance of a Cell processor?

- Single precision (float)
  - 3.2 GHz clock on SPE
  - 4-way SIMD
  - Fused Multiply and Add (FMA)
  - 1 FMA issued every cycle
  - ...adds up to 3.2 * 4 * 2 * 1 = 25.6 GFLOP/s
  - ...but we have 8 SPEs on a Cell processor → 204.8 GFLOP/s
  - ...and we also have a PPE → 230.4 GFLOP/s
- The single precision performance is better than Intel and AMD by an order of magnitude

What is the peak performance of a Cell processor?

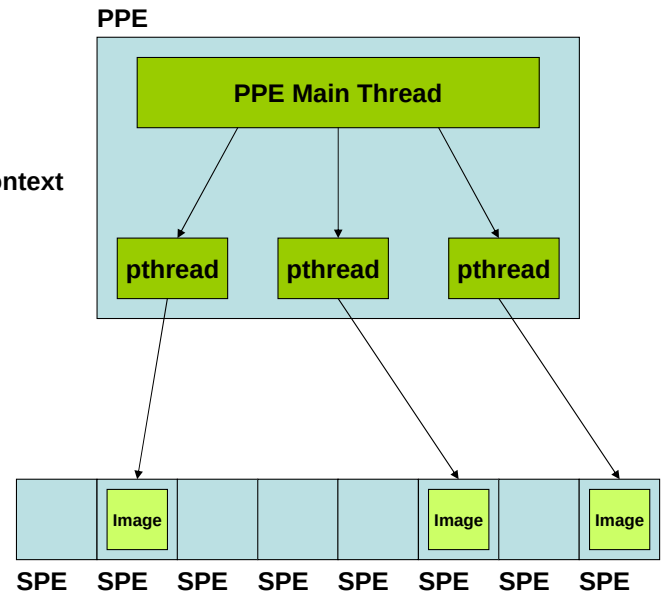
- Double precision (double)
  - 3.2 GHz clock on SPE
  - 2-way SIMD
  - Fused Multiply and Add (FMA)
  - 1 FMA issued every 7 cycles
  - ...adds up to 3.2 * 2 * 2 / 7= 1.83 GFLOP/s
  - ...but we have 8 SPEs on a Cell processor → 14.6 GFLOP/s
  - ...and we also have a PPE → 21.03 GFLOP/s
- The double precision performance is good and a new hardware version makes it better by fully pipelining the double precision instruction execution

Managing the SPEs

libspe2, pthreads, SPE contexts, images

The Overall Picture

Execute PPE program
Create N pthreads
Each pthread:
  Creates SPE context
  Loads program into context
  Runs context
  Destroys context
Join each pthread



PPE Main Thread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>                        // For threads
#include <libspe2.h>                        // For SPE management (spe_*)

void *spe_test_function(void *data);        // pthread callback
extern spe_program_handle_t test_handle;    // Embedded SPE program

int main()
{
   pthread_t my_thread;

   // Create thread
   pthread_create(&my_thread, NULL, spe_test_function, NULL);

   // Join thread
   pthread_join(my_thread, NULL);

   return 0;
}
```

SPE Program

```
#include <stdio.h>

int main(unsigned long long spe_id, unsigned long long pdata)
{
   printf("Hello, Cell!");
   return 0;
}
```

PPE SPE Thread

```
void *spe_test_function(void *data)
{
   int                retval;
   unsigned int       entry_point = SPE_DEFAULT_ENTRY;
   spe_context_ptr_t  my_context;

   /* Create the SPE context */
   my_context = spe_context_create(SPE_EVENTS_ENABLE | SPE_MAP_PS, NULL);

   /* Load the embedded code into this context */
   spe_program_load(my_context, &test_handle);

   /* Run the SPE program until completion */
   do {
      retval = spe_context_run(my_context, &entry_point, 0, NULL, NULL, NULL);
   } while( retval > 0 );

   /* Destroy the context */
   spe_context_destroy(my_context);

   pthread_exit(NULL);
}
```

argp

envp

```
int main(unsigned long long spe_id, unsigned long long pdata)
```

SPE Side

Compiling, linking, executing

```
# Compiling and linking SPE source into SPE executable
spu-gcc  –o  spe_example  spe_example.c

# Embedding SPE executable in PPE object
ppu-embedspu  test_handle  spe_example  spe_example_csf.o

# Compiling and linking PPE executable
ppu-gcc  -o  ppe_example  ppe_example.c  spe_example_csf.o  -lspe2

# Executing
./ppe_example
>> Hello, Cell!
```

SIMD Programming

SIMD

- What is SIMD?
  - Single Instruction Multiple Data
    - One instruction (add, multiply, shift, ...) applied to multiple data
- Registers divided into several logical items
  - 128-bit (2 x 64-bit, 4 x 32-bit, 8 x 16-bit, 16 x 8-bit) on the Cell
- A single instruction operates on all items at once and separate hardware adds parallelism
- On the SPU:
  - **`c = spu_add(a, b)`**
    - Adds the elements of (short) vectors a, b and stores the result in c
  - **`c = spu_madd(a, b, c)`**
    - Multiplies a with b, adds to c and stores result in c

Terminology

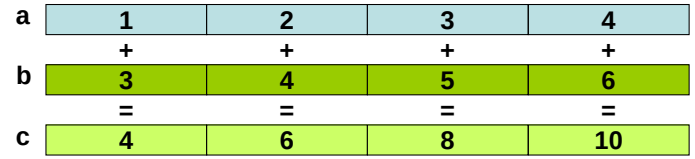
- These terms refer to different numbers of bits:
  - Byte: 8-bit
  - Halfword: 16-bit
  - Word: 32-bit
  - Doubleword: 64-bit
  - Quadword: 128-bit

SIMD Vector Types (C/C++ Extensions)

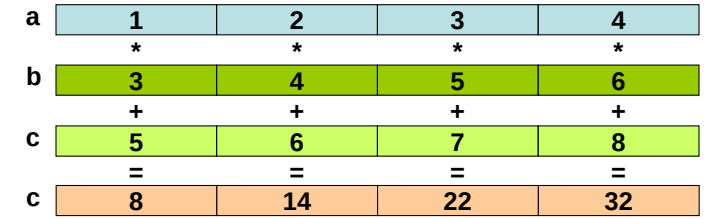
- To allow high-level language programming new data types are defined on the SPU (and PPU) together with compiler intrinsics (i.e., something that looks and functions like a function but is really mapped to a few assembler instructions extremely efficiently).
- These types are available (among others) by including <spu_intrinsics.h>
  - **`vector double (2 x 64-bit)`**
  - **`vector float (4 x 32-bit)`**
  - **`vector unsigned/signed int (4 x 32-bit)`**
  - **`vector unsigned/signed short (8 x 16-bit)`**
  - **`vector unsigned/signed char (16 x 8-bit)`**

SIMD Arithmetic

- **`c = spu_add(a, b)`**

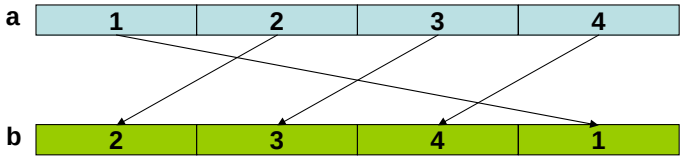


- **`c = spu_madd(a, b, c)`**



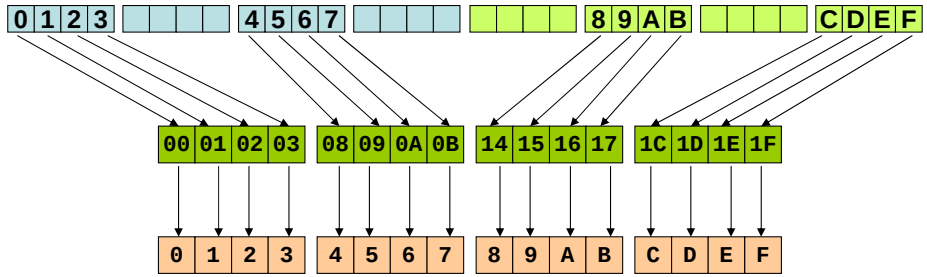
SIMD Rotation

- **`b = spu_rlqwbyte(a, 4)`**
  - Rotate a left 4 bytes (=1 word) and store in b

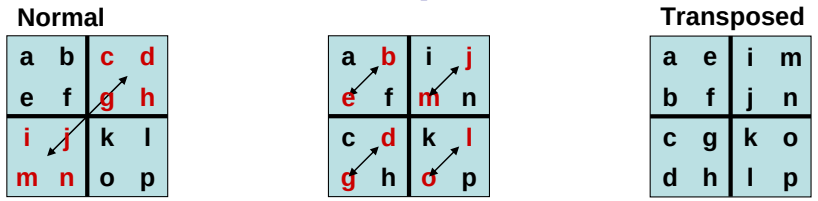


SIMD Shuffle

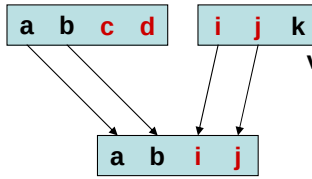
- **`c = spu_shuffle(a, b, pattern)`**
  - Shuffles a and b by selecting bytes and storing in c
  - Each byte in pattern selects a byte in a or b (or one of three constant values)
  - The bytes in a are numbered 0x00 thru 0x0F
  - The bytes in b are numbered 0x10 thru 0x1F
  - This is a very flexible instruction!



SIMD Shuffle example usage: 4x4 Matrix Transpose



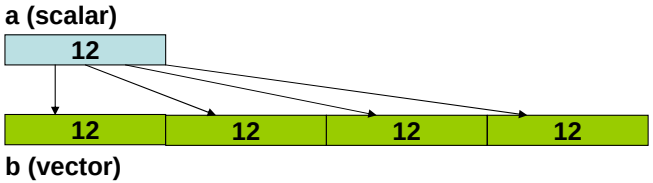
- Can be accomplished using 8 shuffles (and nothing else)
- Example below (creates first row in middle matrix) (constructing all 8 shuffles is left as homework!)



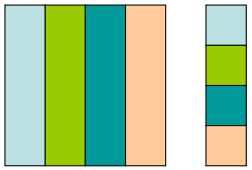

```
vector unsigned byte pat =
   (vector unsigned byte)
      {0x00, 0x01, 0x02, 0x03,
       0x04, 0x05, 0x06, 0x07,
       0x10, 0x11, 0x12, 0x13,
       0x14, 0x15, 0x16, 0x17};
abij = spu_shuffle(abcd, ijkl, pat);
```

SIMD Splat

- Reproduce a scalar into every element of a vector
- `b = spu_splats((unsigned int) 12)`



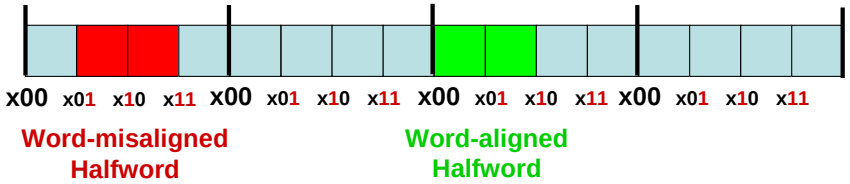
4x4 matrix times 4-vector



```
y = spu_madd(a0, spu_splats(x[0]), y)
y = spu_madd(a1, spu_splats(x[1]), y)
y = spu_madd(a2, spu_splats(x[2]), y)
y = spu_madd(a3, spu_splats(x[3]), y)
```

Alignment

- Alignment refers to how the address of something is aligned to natural borders.
- Addresses are always byte-aligned (since this is the resolution of the address)
- Halfword-aligned addresses have the 1 least significant bit equal to zero
- Word-aligned addresses have the 2 least significant bits equal to zero
- Quadword-aligned addresses have the 4 least significant bits equal to zero



Alignment

- Alignment is crucial since the SPU can only load and store aligned quadwords
- If you try to load (by using assembler for instance) a misaligned address the hardware will adjust your address by ignoring the 4 least significant bits and the data you load will not be what you expect
- If you use C/C++ for instance, the compiler constructs all load and store instructions.
- If you try to load a misaligned item the compiler will use rotations and other instructions to align it properly.
  - This makes load/store of scalars much much slower than for vectors.

Specifying Alignment

- You specify alignment by a compiler directive on the declaration of variables:
  - `float A[16] __attribute__((aligned(16)));`
  - `unsigned int my_int __attribute__((aligned(8)));`
- Argument X tells compiler to align to natural X-byte boundary.
  - `__attribute__((aligned(16)))` // Quadword-aligned

ILP and Loop Optimizations

A very brief review of computer architecture

Instruction Level Parallelism (ILP)

- ”Sequential” processors have for a long time parallelized your applications using hardware that operates on the instruction level (hence; instruction level parallelism)
- ILP techniques:
  - Pipelining
    - Divide instructions into stages and do different stages of several instructions at once.
  - Wide pipelining (SuperScalar processors)
    - Use more than one pipeline to schedule instructions.
  - Out-of-order execution
    - Reorder the instructions to avoid dependency stalls
  - Register renaming
    - ”Rename” registers to remove anti- and output-dependencies
  - Branch prediction
    - Track branch history to reduce pipeline flushes
  - .....

Instruction Level Parallelism (ILP)

- Problem 1:
  - There is not much ILP available in binaries.
- Problem 2:
  - The compiler can often not expose enough ILP to saturate the processor.
- Problem 3:
  - The hardware cost of expanding known features grows rapidly.
- Problem 4:
  - There is a tradeoff between adding deeper pipelines (faster clock) and power, heat dissipation, and pipeline flush penalties.
- Proposed solution (Cell BE):
  - Make the architecture simpler
  - Add more cores instead
  - Forces the programmer to explicitly think about parallelism

Instruction Level Parallelism (ILP)

- Even though the SPEs have only two shallow pipelines, in-order execution and lots of registers, the programmer must sometimes get involved to speed things up.
- We will now discuss a simple example application and introduce a few loop optimizations that optimizing compilers often apply but they can also be used manually.
- The techniques that will be discussed are:
  - Loop unrolling
  - SIMD'zation
  - Software pipelining
- There are many more low-level optimizations that are just as important but are typically managed well by compilers. These techniques include:
  - Register allocation
  - Strength reduction
  - Invariant code motion
  - Instruction scheduling
  - Induction variable elimination
  - .....

Optimization of array multiplication

- Consider the MATLAB command (elementwise multiply):
  - `>> c = a.*b`
  - Where a, b, c are single precision vectors of length N
- We want to make this operation run fast on an SPE

- `void vmul(int N, float *a, float *b, float *c);`

0.13 GFLOP/s
1.00x

Version 0

```
void vmul0(int N, float *a, float *b, float *c)
{
   int i;

   for( i = 0; i < N; i++ ) {
      c[i] = a[i] * b[i];
   }
}
```

Loop Unrolling

- Loop unrolling is when the body of a loop is replicated X times and the iteration step is increased by X.

```
BEFORE:
for( i = 0; i < N; i++ ) {
   c[i] = a[i] * b[i];
}

AFTER:
N3 = N % 3;
for( i = 0; i < N3; i++ ) {  // Clean-up loop
   c[i] = a[i] * b[i];
}
for( i = N3; i < N; i+=3 ) { // Unrolled loop (X=3)
   c[i+0] = a[i+0] * b[i+0];
   c[i+1] = a[i+1] * b[i+1];
   c[i+2] = a[i+2] * b[i+2];
}
```

0.14 GFLOP/s
1.06x

Version 1

Unrolling (4)

```
void vmul1(int N, float *a, float *b, float *c)
{
   int i;

   for( i = 0; i < N; i+=4 ) {
      c[i+0] = a[i+0] * b[i+0];
      c[i+1] = a[i+1] * b[i+1];
      c[i+2] = a[i+2] * b[i+2];
      c[i+3] = a[i+3] * b[i+3];
   }
}
```

SIMD'zation

- SIMD'zation is when we replace several scalar instructions by one or more SIMD instructions. Often we may have to change the data structures to improve the ease of SIMD'zation (not so in this example though).

```
BEFORE:
for( i = 0; i < N; i+=4 ) {
   c[i+0] = a[i+0] * b[i+0];
   c[i+1] = a[i+1] * b[i+1];
   c[i+2] = a[i+2] * b[i+2];
   c[i+3] = a[i+3] * b[i+3];
}

AFTER:
vector unsigned int *va = (vector unsigned int*)&a[0];
vector unsigned int *vb = (vector unsigned int*)&b[0];
vector unsigned int *vc = (vector unsigned int*)&c[0];
for( i = 0; i < N/4; i++ ) {
   vc[i] = spu_mul(va[i], vb[i]);
}
```

0.80 GFLOP/s
6.00x

Version 2

Why do we get > 4x speedup?

Unrolling (4)
SIMD'zation

```
void vmul2(int N, float *a, float *b, float *c)
{
   int i;
   vector float *va = (vector float *)a,
                *vb = (vector float *)b,
                *vc = (vector float *)c;
   int N4 = N/4;

   for( i = 0; i < N4; i++ ) {
      vc[i] = spu_mul(va[i], vb[i]);
   }
}
```

Explicit Load and Store

- In order to make use of more advanced optimization techniques we need to make loads and stores explicit in our C program.
- Consider the C statement: `vc[i] = spu_mul(va[i], vb[i]);`
  - This is broken down (by the compiler) into:
    - `load va[i]`
    - `load vb[i]`
    - `compute spu_mul()`
    - `store vc[i]`
  - We can make this explicit in our C program as such:

```
vai = va[i];              // Load va[i]
vbi = vb[i];              // Load vb[i]
vci = spu_mul(vai, vbi); // Compute spu_mul()
vc[i] = vci;              // Store vc[i]
```

  - If we declare vai, vbi, vci as local variables the compiler will be clever enough to map these into registers.

0.80 GFLOP/s
6.00x

Version 3

(no gain, but enables us to do other things)

Unrolling (4)
SIMD'zation
Explicit load/store

```
void vmul3(int N, float *a, float *b, float *c)
{
   int i;
   vector float *va = (vector float *)a,
                *vb = (vector float *)b,
                *vc = (vector float *)c;
   vector float ra, rb, rc;
   int N4 = N/4;

   for( i = 0; i < N4; i++ ) {
      ra = va[i];              // Load
      rb = vb[i];              // ...
      rc = spu_mul(ra, rb);    // Compute
      vc[i] = rc;              // Store
   }
}
```

Software Pipelining

- A pipeline has S stages and executes S different "instructions" concurrently by executing a different piece of each every cycle.
- Doing the same with a loop is called Software Pipelining
- Care must be taken not to break dependencies

Software Pipelining

```
BEFORE:
for( i = 0; i < 8; i++ ) {
   // Stage A (i)
   // Stage B (i)
   // Stage C (i)
}

AFTER:
/* Prologue */
// Stage A (0)
// Stage B (0)
// Stage A (1)
/* Steady state loop */
for( i = 0; i < 6; i++ ) {
   // Stage C (i+0)
   // Stage B (i+1)
   // Stage A (i+2)
}
/* Epilogue */
// Stage C (6)
// Stage B (7)
// Stage C (7)
```

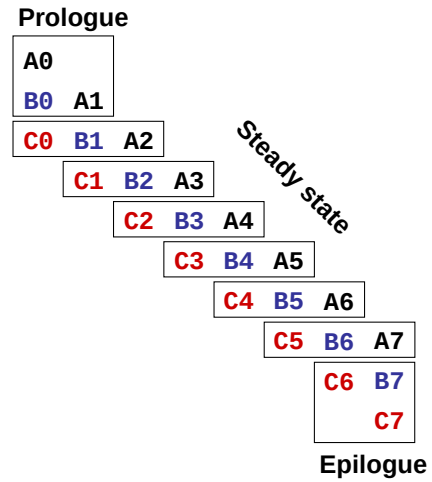
Software Pipelining

BEFORE

Steady state

A0	A1	A2	A3	A4	A5	A6	A7
B0	B1	B2	B3	B4	B5	B6	B7
C0	C1	C2	C3	C4	C5	C6	C7

AFTER



1.59 GFLOP/s
11.95x

Version 4

Unrolling (4)
SIMD'zation
Explicit load/store
SW Pipelining

```
void vmul4(int N, float *a, float *b, float *c)
{
   int i;
   vector float *va = (vector float *)a,
                *vb = (vector float *)b,
                *vc = (vector float *)c;
   vector float ra, rb, rc;
   int N4 = N/4;

   /* Prologue */
   ra = va[0];                 // Load     [0]
   rb = vb[0];                 // ...
   rc = spu_mul(ra, rb);       // Compute  [0]
   ra = va[0];                 // Load     [1]
   rb = vb[0];                 // ...
   /* Steady state loop */
   for( i = 0; i < N4-2; i++ ) {
      vc[i+0] = rc;            // Store    [i]
      rc = spu_mul(ra, rb);    // Compute  [i+1]
      ra = va[i+2];            // Load     [i+2]
      rb = vb[i+2];            // ...
   }
   /* Epilogue */
   vc[N4-2] = rc;              // Store    [N4-2]
   rc = spu_mul(ra, rb);       // Compute  [N4-1]
   vc[N4-1] = rc;              // Store    [N4-1]
}
```

1.82 GFLOP/s
13.64x

Version 5

Unrolling (4)
SIMD'zation
Explicit load/store
SW Pipelining
Unrolling (2)

```
void vmul5(int N, float *a, float *b, float *c)
{
   int i;
   vector float *va = (vector float *)a,
                *vb = (vector float *)b,
                *vc = (vector float *)c;
   vector float ra, rb, rc;
   int N4 = N/4;

   /* Prologue */
   /* --- snip --- */
   /* Steady state loop */
   for( i = 0; i < N4-2; i+=2 ) {
      vc[i+0] = rc;            // Store    [i]
      rc = spu_mul(ra, rb);    // Compute  [i+1]
      ra = va[i+0+2];          // Load     [i+2]
      rb = vb[i+0+2];          // ...

      vc[i+1] = rc;            // Store    [i+1]
      rc = spu_mul(ra, rb);    // Compute  [i+2]
      ra = va[i+1+2];          // Load     [i+3]
      rb = vb[i+1+2];          // ...
   }
   /* Epilogue */
   /* --- snip --- */
}
```

2.82 GFLOP/s
21.17x

Version 6

Unrolling (4)
SIMD'zation
Explicit load/store
Unrolling (2)
SW Pipelining

```
void vmul6(int N, float *a, float *b, float *c)
{
   int i;
   vector float *va = (vector float *)a,
                *vb = (vector float *)b,
                *vc = (vector float *)c;
   vector float ra0, ra1, rb0, rb1, rc0, rc1;
   int N4 = N/4;

   /* Prologue */
   /* --- snip --- */
   /* Steady state loop */
   for( i = 0; i < N4-2; i+=2 ) {
      vc[i+0+0] = rc0;          // Store   [i+0]
      vc[i+0+1] = rc1;          // Store   [i+1]
      rc0 = spu_mul(ra0, rb0);  // Compute [i+2]
      rc1 = spu_mul(ra1, rb1);  // Compute [i+3]
      ra0 = va[i+4+0];          // Load    [i+4]
      ra1 = va[i+4+1];          // Load    [i+5]
      rb0 = vb[i+4+0];          // Load    [i+4]
      rb1 = vb[i+4+1];          // Load    [i+5]
   }
   /* Epilogue */
   /* --- snip --- */
}
```

What is peak for this operation?

- N floating point operations
- 2 load + 1 store per 4 iterations → 3*N/4 mem. ops.
- SPE issues 1 mem. op. per cycle → > C ≡ 3*N/4 cycles
- 3.2 GHz clock on SPE → < N*3.2/C = 4.27 GFLOP/s

- Version 6: 2.82 GFLOP/s (~66% of peak)

- Floating point peak of SPE: 25.6 GFLOP/s
- Why are we so far from peak?
  - No FMA (0.5) → 12.8 GFLOP/s
  - 3 mem.op. per multiply (0.33) → 4.27 GFLOP/s

Conclusions

- Even on an architecture with low latency memory and memory bandwidth equal to four words per cycle the memory wall is still a big issue.
- Try to find new algorithms were the computational density (flop/mem.op.) is larger to reduce the memory wall impact.

Additional Techniques: Blocking

- Loop tiling (blocking) is a ubiquitous optimization used primarily to improve data reuse (in caches or the register file).
- It is the multi-dimensional equivalent of strip mining.

```
BEFORE:
for( i = 0; i < N; i++ )
   for( j = 0; j < N; j++ )
      a[i + j*N] = b[j + i*N]; // Stride-N on A
                               // Stride-1 on B

AFTER:
for( ii = 0; ii < N; ii += 64 )
   for( jj = 0; jj < N; jj += 64 )
      for( i = ii; i < min(ii+64, N); i++ )
         for( j = jj; j < min(jj+64, N); j++ )
            a[i + j*N] = b[j + i*N];
```

Lab 4:
SGEMM on Cell

Optimization Techniques

Storage Schemes

- When you store a 2D matrix in contiguous storage you must map the elements to linear memory.
- Can be done in many many many different ways.
- Two common, canonical, schemes (M x N matrix, zero-based indexing):
  - Row Major (pack rows after each other)
    - A(i, j) maps to mem(i*N + j)
  - Column Major (pack columns on top of each other)
    - A(i, j) maps to mem(i + j*M)

Strided Access

```
for( i = 0; i < M*N; i++ )
   a[i] = ...;   // Stride-1

for( i = 0; i < N; i++ )
   a[i + j*M] = ...; // Stride-1

for( j = 0; j < N; j++ )
   a[i + j*M] = ...; // Stride-M

for( k = 0; k < K; k++ )
   ... += a[i + k*M] * b[k + j*M]; // Stride-M on a
                                   // Stride-1 on b
```

- Stride-1 access is what we strive for:
  - On cache-based system this gives us good data locality.
  - On SIMD architectures we can load a vector easily.

GEMM Transposition Patterns

```
N, N: A * B
N, T: A * B^T
T, N: A^T * B
T, T: A^T * B^T
```

- We have four transposition patterns for GEMM.
- This will impact how to implement GEMM efficiently.
- Implementation options also depend on the storage scheme.

- In the assignment, you will look at one case only:
  - T, N transposition pattern (A^T * B)
  - Column major storage
  - N x N matrices (M = K = N)
  - N = 64

GEMM Loop Order

```
NAIVE IMPLEMENTATION OF T,N GEMM AND COLUMN MAJOR
(DOT PRODUCT FORMULATION)

for( j = 0; j < N; j++ )                    // Columns of C
   for( i = 0; i < N; i++ )                 // Rows of C
      for( k = 0; k < N; k++ )              // Inner loop
         C[i + j*N] += A[k + i*N] * B[k + j*N]; // Stride-1 on A and B
```

Loop order: j, i, k

- The three loops can be ordered in 6 different ways.
- They all have pros and cons.
- Dot product formulation (above) has the following pros:
  - Stride-1 access on A and B in inner loop
  - Stride-1 access on C in middle loop
  - Load and stores in inner loop (few compared with other variants):
    - 1 scalar load
    - 1 scalar store
    - 2 vector loads

High Precision Time Measurements

- In order to make high resolution time measurements on the SPU the designers have added a 32-bit decrementer register that can be read and written by user applications using special assembler instructions.
- QS21 decrementer frequency: 26.67 MHz (/proc/cpuinfo)
- One tick equals 120 cycles.
- Decrementer underflows after 161 seconds.
- Snippet to measure some small piece (< 161 seconds) of code:

```
uint32_t t1, t2;
spu_write_decrementer(0xFFFFFFFF); // Initialize
t1 = spu_read_decrementer();       // Start
// DO SOME WORK
t2 = spu_read_decrementer();       // Finish
ticks = t1 – t2
seconds_QS21 = ((float) ticks)/26666666;
```

What you will be given

- You will be given a working skeleton code which computes C = C + A^T*B where A, B, C are 64x64 contiguous float matrices.
- The PPE code creates matrices, launches an SPE program to compute SGEMM and checks the result and computes performance.
- The SPE SGEMM implementation is the naive dot product formulation (j, i, k) variant and when compiled with optimization level 3 runs at
  - 0.84% of peak (0.215 GFLOP/s)
- Without any compiler flags:
  - 0.15% of peak (0.038 GFLOP/s)
- Conclusion: compiler improves performance 5.7x
  - If we feel lucky and remove volatile keyword compiler improves 12x.

Your task

- Your task:
  - Optimize the SPE SGEMM implementation using, for example:
    - Loop unrolling,
    - SIMD'zation,
    - Software pipelining,
    - Loop linearization,
    - Blocking.
- Restrictions:
  - Change only the SGEMM function in spe.c
  - To pass the assignment, performance must be > 5 GFLOP/s