# Cell Broadband Engine
# Optimization and Programming Models

Lars Karlsson

May 5, 2009

# Part I

# Optimization

# Things to Think About...

- instruction count
- instruction latencies
- instruction mix
- DMA transfers
- branch mispredictions
- pipeline stalls

# Loop Unrolling

- Loop unrolling duplicates the loop body several times.

```
// Before:
for( int i = 0; i < N; i++ ) {
   a[i] *= 2;
}
// After:
for( int i = 0; i < N; i += 4 ) {
   a[i+0] *= 2;
   a[i+1] *= 2;
   a[i+2] *= 2;
   a[i+3] *= 2;
}
```

- Care must be used to handle cases when the number of iterations is not a multiple of the loop unrolling factor.
- Benefits:
  - Increases size of loop body.
  - Reduces loop branching overhead.

# Branch Elimination (1)

- Unrolling: remove loop branching.

```
// Before:
for( int i = 0; i < 4; i++ ) {
   c[i] = a[i] * b[i];
}
// After:
c[0] = a[0] * b[0];
c[1] = a[1] * b[1];
c[2] = a[2] * b[2];
c[3] = a[3] * b[3];
```

# Branch Elimination (2)

- Inlining: remove function call branching.

```
// Before:
vector float foo(vector float v) {
   return v * spu_splats((float) 2);
}
x = foo(x);
// After:
x = x * spu_splats((float) 2);
```

# Branch Elimination (3)

- Predication: avoid if..then..else by speculative computation.

```
// Before:
if( a[i] < 0 )
   a[i] = 0;
else
   a[i] *= 2;
// After:
mask = spu_cmpgt(-a, 0); // check condition
res_true  = spu_splats((float) 0);     // if-clause
res_false = a * spu_splats((float) 2); // else-clause
res = spu_sel(res_false, res_true, mask); // combine results
```

# Loop Peeling

- Loop peeling can be used to move border cases outside a loop.

  ```
  for( int i = 0; i < N; i++ ) {
     if( i == 0 ) a[i] += N;
     else a[i] -= 1;
  }
  ```

  becomes

  ```
  if( 0 < N ) a[0] += N;
  for( int i = 1; i < N; i++ ) {
     a[i] -= 1;
  }
  ```

- Benefits:
  - Eliminates branching inside loop.
  - Simplifies loop body.

# Loop Fusion

- Loop fusion collapses two loops with the same iteration space.

```
for( int i = 0; i < N; i++ ) {
   a[i] *= 2;
}
for( int i = 0; i < N; i++ ) {
   b[i] -= 5;
}
```

  becomes

```
for( int i = 0; i < N; i++ ) {
   a[i] *= 2;
   b[i] -= 5;
}
```

- Benefits:
  - Reduces loop overhead (fewer loops).
  - Increases size of loop body.

# Software Pipelining

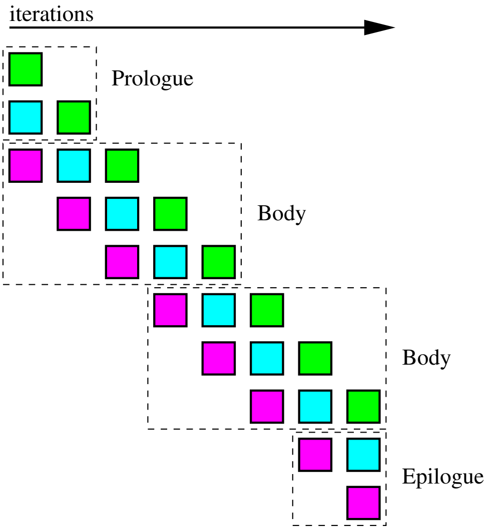
- Loop body consisting of phases that are sequentially dependent:

```
for( int i = 0; i < N; i++ ) {
   A(i);
   B(i);
   C(i);
}
```

- Rearrange loop to do A, B, C from different iterations:

```
A(0);
B(0); A(1);
for( int i = 0; i < N-2; i++ ) {
   C(i+0);
   B(i+1);
   A(i+2);
}
C(i+0); B(i+1);
        C(i+1);
```

# Software Pipelining: Illustration



# Part II

## Programming Models

# Function-Offload Model

- Also known as the Remote Procedure Call (RPC) Model.
- SPEs accelerate performance critical procedures.
- Quickest way to take advantage of SPEs within existing application.
- Main application runs on PPE and calls selected procedures on one or more SPEs.
- Method stubs on PPE and SPE handle data transfers and synchronization.

# Device-Extension Model

- Special case of Function-Offload Model where SPEs act like I/O devices.
- All I/O devices are memory mapped so the SPEs can interact with them.
- Usually runs priviliged code which is part of the operating system.
- E.g., encrypted file system: SPE used to offload encryption/decryption.

# Computation-Acceleration Model

- SPE-centric model which enables finer granularity.
- Speeds up applications that use computation-intensive mathematical functions.
- Most computation performed by SPEs in parallel.
- Computation partitioned manually.
- PPE acts as control and system management hub.
- Can use either shared memory or message passing to communicate between SPEs.

# Streaming Model

- Each SPE computes on data that streams through it.
- The PPE acts as a stream controller and the SPEs as stream data processors.
- SPEs organized in pipeline fashion enables effective use of the high on-chip bandwidth.
- Double buffering hides communication overhead.

# Shared-Memory Multiprocessor Model

- DMA transfers are cache-coherent and all units have access to system memory.
- Shared memory read replaced by DMA to local store followed by read to register.
- Shared memory store replaced by write to local store followed by DMA to main memory.
- Synchronization via atomic operations or higher level objects such as mutexes and condition variables.

# Assymetric-Thread Runtime Model

- Thread run on either PPE or SPE.
- Threads interact the same way they do in a conventional symmetric multiprocessor.
- Flexible model which supports all of the other models.
- This is the fundamental model provided by the SDK.

# SPE Overlays

- When code does not fit in an SPEs local store, overlays can be used.
- Several code sections share the same memory space.
- Stubs load the required code dynamically.
- Linker assists in creating overlays.