Performance

Main topics:

- Performance analysis
- Performance issues
- Static analysis of SPE threads
- Dynamic analysis of SPE threads
- Optimizations
- Static analysis of optimization
- Dynamic analysis of optimizations
- General SPE programming tips

Performance analysis

After a Cell Broadband Engine program executes without errors on the PPE and the SPEs, optimization through parameter-tuning can begin.

Programmers typically tune for performance using algorithmic methods. This is important for SPE programming also. But equally important for SPE programming is performance tuning through the elimination of stalls. There are two forms of stalls to consider:

- instruction dependency stalls, and
- data stalls.

Instruction stalls can be analyzed statically or dynamically. Two software tools are available in the SDK to assist in measuring the performance of programs: the *spu-timing* static timing analyzer, and the *IBM Full System Simulator for the Cell Broadband Engine*.

Performance issues

Two software tools are available in the SDK to assist in measuring the performance of programs: the *spu-timing* static timing analyzer, and the *IBM Full System Simulator for the Cell Broadband Engine*.

The *spu-timing* analyzer performs a static timing analysis of a program by annotating its assembly instructions with the instruction-pipeline state. This analysis is useful for coarsely spotting dual-issue rates (odd and even pipeline use) and assessing what program sections may be experiencing instruction-dependency and data-dependency stalls. It is useful, for example, for determining whether or not dependencies might be mitigated by unrolling, or whether reordering of instructions or better placement of no-ops will improve the dual-issue behavior in a loop. However, static analysis outputs typically do not provide numerical performance information about program execution. Thus, it cannot report anything definitive about cycle counts, branches taken or not taken, branches hinted or not hinted, DMA transfers, and so forth.

The *IBM Full System Simulator for the Cell Broadband Engine* performs a dynamic analysis of program execution. It is available in the SDK. Any part of a program, from a single line to the entire program, can be studied. Performance numbers are provided for:

- Instruction histograms (for example, branch, hint, and prefetch)
- Cycles per instruction (CPI)
- Single-issue and dual-issue rates
- Stall statistics
- Register use

The output of the IBM Full System Simulator for the Cell Broadband Engine can be a text listing or a graphic plot.

Static analysis of SPE threads

The listing below shows an `spu-timing` static timing analysis for the inner loop of the SPE code.

The SPE code show below is from the Euler Particle-System Simulation example.

SPE Code:

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#define PARTICLES_PER_BLOCK             1024
// Local store structures and buffers.
volatile context ctx;
volatile vector float pos[PARTICLES_PER_BLOCK];
volatile vector float vel[PARTICLES_PER_BLOCK];
volatile float inv_mass[PARTICLES_PER_BLOCK];
int main(unsigned long long spe_id, unsigned long long parm)
{
  int i, j;
  int left, cnt;
  float time;
  unsigned int tag_id;
  vector float dt_v, dt_inv_mass_v;
        /* Reserve a tag ID */
  tag_id = mfc_tag_reserve();
  spu_writech(MFC_WrTagMask, -1);
  // Input parameter parm is a pointer to the particle context.
  // Fetch the context, waiting for it to complete.
  spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context),
    tag_id, MFC_GET_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  dt_v = spu_splats(ctx.dt);
  // For each step in time
  for (time=0; time<END_OF_TIME; time += ctx.dt) {
    // For each block of particles
    for (i=0; i<ctx.particles; i+=PARTICLES_PER_BLOCK) {
      // Determine the number of particles in this block.
      left = ctx.particles - i;
      cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;
      // Fetch the data - position, velocity, inverse_mass. Wait for DMA to
      // complete before performing computation.
      spu_mfcdma32((void *)(pos), (unsigned int)(ctx.pos_v+i), cnt *
        sizeof(vector float), tag_id, MFC_GET_CMD);
      spu_mfcdma32((void *)(vel), (unsigned int)(ctx.vel_v+i), cnt *
        sizeof(vector float), tag_id, MFC_GET_CMD);
      spu_mfcdma32((void *)(inv_mass), (unsigned int)(ctx.inv_mass+i), cnt *
        sizeof(float), tag_id, MFC_GET_CMD);
      (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

```
      // Compute the step in time for the block of particles
      for (j=0; j<cnt; j++) {
        pos[j] = spu_madd(vel[j], dt_v, pos[j]);
        dt_inv_mass_v = spu_mul(dt_v, spu_splats(inv_mass[j]));
        vel[j] = spu_madd(dt_inv_mass_v, ctx.force_v, vel[j]);
      }
      // Put the position and velocity data back into main storage
      spu_mfcdma32((void *)(pos), (unsigned int)(ctx.pos_v+i), cnt *
        sizeof(vector float), tag_id, MFC_PUT_CMD);
      spu_mfcdma32((void *)(vel), (unsigned int)(ctx.vel_v+i), cnt *
        sizeof(vector float), tag_id, MFC_PUT_CMD);
    }
  }
  // Wait for final DMAs to complete before terminating SPE thread.
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  return (0);
}
```

The following listing shows significant dependency stalls (indicated by the "-") and poor dual-issue rates. The inner loop has an instruction mix of eight even-pipeline (pipe 0) instructions and ten odd-pipeline (pipe 1) instructions. Therefore, any program changes that minimize data dependencies will improve dual-issue rates and lower the cycle per instruction (CPI).

```
                                                        .L19:
0D                                                 78       a       $49,$8,$10
1D 012                                             789      lqx     $51,$6,$9
0D                                                  89      ila     $47,66051
1D 0123                                             89      lqx     $52,$6,$11
0  0                                                 9      ai      $7,$7,-1
0  ----456789                                              fma     $50,$51,$12,$52
1       -----012345                                        stqx    $50,$6,$11
1            123456                                        lqx     $48,$8,$10
0D            23                                           ai      $8,$8,4
1D            234567                                       lqa     $44,ctx+16
1              345678                                      lqx     $43,$6,$9
1               ---7890                                    rotqby  $46,$48,$49
1                   ---1234                                shufb   $45,$46,$46,$47
0                       ---567890                          fm      $42,$12,$45
0d                          -----123456                    fma     $41,$42,$44,$43
1d                               ------789012               stqx    $41,$6,$9
0D                                     89                   ai      $6,$6,16
                                                       .L39:
1D                                     8901                 brnz    $7,.L19
```

The character columns in the above static-analysis listing have the following meanings:

- Column 1: The first column shows the pipeline that issued an instruction. Pipeline 0 is represented by `0` in the first column and pipeline 1 is represented by `1`.
- Column 2: The second column can contain a `D`, `d`, or "nothing". A `D` signifies a successful dual-issue was accomplished by the two instructions listed in row-pairs. A `d` signifies a dual-issue was possible, but did not occur due to dependencies; for example, operands being in flight. If there is no entry in the second column, dual-issue could not be performed because the issue rules were not satisfied (for example, an even-pipeline instruction was fetched from an odd LS address or an odd-pipeline instruction was fetched from an even LS address). Read about pipelines and dual-issue rules.
- Column 3: The third column is always blank.
- Columns 4 through 53: The next 50 columns represent clock cycles and are repeated as `0123456789` five times. A digit is displayed in these columns whenever the instruction executes during that clock cycle. Therefore, an `<n>-cycle` instruction will display `<n>` digits. Dependency stalls are flagged by a dash ("-").
- Columns 54 and beyond: The remaining entries on the row are the assembly-language

instructions or assembler-line addresses (for example, `.L19`) of the program's assembly code.

Static-analysis timing files can be quickly interpreted by:

- Scanning the columns of digits. Small slopes (more horizontal) are bad. Large slopes (more vertical) are good.
- Looking for instructions with dependencies (those with dashes in the listing).
- Looking for instructions with poor dual-issue rates: either a `d` or "nothing" in column 2.

This information can be used to understand what areas of code are scheduled well and which are poorly scheduled.

About SPU_TIMING:

If you are using a Bash shell, you can set `SPU_TIMING` as a shell variable by using the command `export SPU_TIMING=1`. You can also set `SPU_TIMING` in the makefile and build the `.s` file by using the following statement:

```
SPU_TIMING=1 make foo.s
```

This creates the timing file for file `foo.c` . It sets the `SPU_TIMING` variable only in the sub-shell of the makefile. It generates `foo.s` and then invokes `spu-timing` on `foo.s` to produce a `foo.s.timing` file.

Another way to invoke the performance tool is by entering one of the following statements in the command prompt:

```
SPU_TIMING=1 make foo.s
```

Dynamic analysis of SPE threads

The listing below shows a dynamic timing analysis on the same SPE inner loop using the *IBM Full System Simulator for the Cell Broadband Engine*.

The results confirm the view of program execution from the static timing analysis:

- It shows poor dual-issue rates (7%) and large dependency stalls (65%), resulting in a overall CPI of 2.39.
- Most workloads should be capable of achieving a CPI of 0.7 to 0.9, roughly 3 times better than this.
- The number of used registers is 73, a 57.03% utilization of the full 128 register set.

```
SPU DD1.0
***
Total Cycle count               43120454
Total Instruction count         18068949
Total CPI                       2.39
***
Performance Cycle count         43120454
Performance Instruction count   18068949 (18062968)
Performance CPI                 2.39 (2.39)

Branch instructions             1001990
Branch taken                    1000007
Branch not taken                1983
```

```
Hint instructions               1973
Hint hit                        1000001

Contention at LS between Load/Store and Prefetch 2000986

Single cycle                                          12049144 ( 27.9%)
Dual cycle                                             3006912 (  7.0%)
Nop cycle                                                 4003 (  0.0%)
Stall due to branch miss                                 17977 (  0.0%)
Stall due to prefetch miss                                   0 (  0.0%)
Stall due to dependency                               28042299 ( 65.0%)
Stall due to fp resource conflict                            0 (  0.0%)
Stall due to waiting for hint target                       110 (  0.0%)
Stall due to dp pipeline                                     0 (  0.0%)
Channel stall cycle                                          0 (  0.0%)
SPU Initialization cycle                                     9 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                           43120454 (100.0%)

Stall cycles due to dependency on each pipelines
 FX2        5909
 SHUF       6011772
 FX3        1960
 LS         7022608
 BR         0
 SPR        0
 LNOP       0
 NOP        0
 FXB        0
 FP6        15000050
 FP7        0
 FPD        0

The number of used registers are 73; the used ratio is 57.03
```

Optimizations

To eliminate stalls and improve the CPI and ultimately the performance the compiler needs more instructions to schedule, so that the program does not stall. The SPE's large register file allows the compiler or the programmer to unroll loops.

In our example program, there are no inter-loop dependencies (loop-carried dependencies), and our dynamic analysis shows that the register usage is fairly small, so moderately aggressive unrolling will not produce register spilling (that is, registers having to be written into temporary stack storage).

Most compilers can automatically unroll loops. Sometimes this is effective. But because automatic loop unrolling is not always effective, or because the programmer wants explicit control to manage the limited local store, this example shows how to manually unroll the loop.

The first pass of optimizations include:

- Unroll the loop to provide additional instructions for interleaving.
- Load DMA-buffer contents into local nonvolatile registers to eliminate volatile migration constraints.
- Eliminate scalar loads (the `inv_mass` variable).
- Eliminate extra multiplies of `dt*inv_mass` and `splat` the products after the SIMD multiply, instead of before the multiply.
- Interleave DMA transfers with computation by multibuffering the inputs and outputs to eliminate

(or reduce) DMA stalls. These stalls are not reflected in the static and dynamic analyses. In the process of adding double buffering, the inner loop is moved into a function, so that the code need not be repeated.

The following SPE code results from these optimizations. Among the changes are the addition of a `GET` instruction with a barrier suffix (`B`), accomplished by the `spu_mfcdma32()` intrinsic with the `MFC_GETB_CMD` parameter. This `GET` is the barrier form of `MFC_GET_CMD`. The barrier form is used to ensure that previously computed results are `put` before the `get` for the next buffer's data.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#define PARTICLES_PER_BLOCK             1024
// Local store structures and buffers.
volatile context ctx;
volatile vector float pos[2][PARTICLES_PER_BLOCK];
volatile vector float vel[2][PARTICLES_PER_BLOCK];
volatile vector float inv_mass[2][PARTICLES_PER_BLOCK/4];
void process_buffer(int buffer, int cnt, vector float dt_v)
{
  int i;
  volatile vector float *p_inv_mass_v;
  vector float force_v, inv_mass_v;
  vector float pos0, pos1, pos2, pos3;
  vector float vel0, vel1, vel2, vel3;
  vector float dt_inv_mass_v, dt_inv_mass_v_0, dt_inv_mass_v_1,
    dt_inv_mass_v_2, dt_inv_mass_v_3;
  vector unsigned char splat_word_0 =
        (vector unsigned char){0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
  vector unsigned char splat_word_1 =
        (vector unsigned char){4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7};
  vector unsigned char splat_word_2 =
        (vector unsigned char){8, 9,10,11, 8, 9,10,11, 8, 9,10,11, 8, 9,10,11};
  vector unsigned char splat_word_3 =
        (vector unsigned char){12,13,14,15,12,13,14,15,12,13,14,15,12,13,14,15};
  p_inv_mass_v = (volatile vector float *)&inv_mass[buffer][0];
  force_v = ctx.force_v;
  // Compute the step in time for the block of particles, four
  // particle at a time.
  for (i=0; i<cnt; i+=4) {
    inv_mass_v = *p_inv_mass_v++;

    pos0 = pos[buffer][i+0];
    pos1 = pos[buffer][i+1];
    pos2 = pos[buffer][i+2];
    pos3 = pos[buffer][i+3];
    vel0 = vel[buffer][i+0];
    vel1 = vel[buffer][i+1];
    vel2 = vel[buffer][i+2];
    vel3 = vel[buffer][i+3];
    dt_inv_mass_v = spu_mul(dt_v, inv_mass_v);
    pos0 = spu_madd(vel0, dt_v, pos0);
    pos1 = spu_madd(vel1, dt_v, pos1);
    pos2 = spu_madd(vel2, dt_v, pos2);
    pos3 = spu_madd(vel3, dt_v, pos3);
    dt_inv_mass_v_0 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_0);
    dt_inv_mass_v_1 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_1);
    dt_inv_mass_v_2 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_2);
    dt_inv_mass_v_3 = spu_shuffle(dt_inv_mass_v, dt_inv_mass_v, splat_word_3);
    vel0 = spu_madd(dt_inv_mass_v_0, force_v, vel0);
    vel1 = spu_madd(dt_inv_mass_v_1, force_v, vel1);
    vel2 = spu_madd(dt_inv_mass_v_2, force_v, vel2);
    vel3 = spu_madd(dt_inv_mass_v_3, force_v, vel3);
```

```
    pos[buffer][i+0] = pos0;
    pos[buffer][i+1] = pos1;
    pos[buffer][i+2] = pos2;
    pos[buffer][i+3] = pos3;
    vel[buffer][i+0] = vel0;
    vel[buffer][i+1] = vel1;
    vel[buffer][i+2] = vel2;
    vel[buffer][i+3] = vel3;
  }
}
int main(unsigned long long spe_id, unsigned long long argv)
{
  int buffer, next_buffer;
  int cnt, next_cnt, left;
  float time, dt;
  vector float dt_v;
  volatile vector float *ctx_pos_v, *ctx_vel_v;
  volatile vector float *next_ctx_pos_v, *next_ctx_vel_v;
  volatile float *ctx_inv_mass, *next_ctx_inv_mass;
  unsigned int tags[2];
  // Reserve a pair of DMA tag IDs
  tags[0] = mfc_tag_reserve();
  tags[1] = mfc_tag_reserve();
  // Input parameter argv is a pointer to the particle context.
  // Fetch the context, waiting for it to complete.
  spu_writech(MFC_WrTagMask, 1 << tags[0]);
  spu_mfcdma32((void *)(&ctx), (unsigned int)argv, sizeof(context), tags[0],
    MFC_GET_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  dt = ctx.dt;
  dt_v = spu_splats(dt);
  // For each step in time
  for (time=0; time<END_OF_TIME; time += dt) {
    // For each double buffered block of particles
    left = ctx.particles;
    cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;
    ctx_pos_v = ctx.pos_v;
    ctx_vel_v = ctx.vel_v;
    ctx_inv_mass = ctx.inv_mass;
    // Prefetch first buffer of input data
    buffer = 0;
    spu_mfcdma32((void *)(pos), (unsigned int)(ctx_pos_v), cnt *
      sizeof(vector float), tags[0], MFC_GETB_CMD);
    spu_mfcdma32((void *)(vel), (unsigned int)(ctx_vel_v), cnt *
      sizeof(vector float), tags[0], MFC_GET_CMD);
    spu_mfcdma32((void *)(inv_mass), (unsigned int)(ctx_inv_mass), cnt *
      sizeof(float), tags[0], MFC_GET_CMD);
    while (cnt < left) {
      left -= cnt;
      next_ctx_pos_v = ctx_pos_v + cnt;
      next_ctx_vel_v = ctx_vel_v + cnt;
      next_ctx_inv_mass = ctx_inv_mass + cnt;
      next_cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;
      // Prefetch next buffer so the data is available for computation on next
      //   loop iteration.
      // The first DMA is barriered so that we don't GET data before the
      //   previous iteration's data is PUT.
      next_buffer = buffer^1;
      spu_mfcdma32((void *)(&pos[next_buffer][0]), (unsigned int)(next_ctx_pos_v),
        next_cnt * sizeof(vector float), tags[next_buffer], MFC_GETB_CMD);
      spu_mfcdma32((void *)(&vel[next_buffer][0]), (unsigned int)(next_ctx_vel_v),
        next_cnt * sizeof(vector float), tags[next_buffer], MFC_GET_CMD);
      spu_mfcdma32((void *)(&inv_mass[next_buffer][0]), (unsigned int)
        (next_ctx_inv_mass), next_cnt * sizeof(float), tags[next_buffer],
```

```
          MFC_GET_CMD);

      // Wait for previously prefetched data
      spu_writech(MFC_WrTagMask, 1 << tags[buffer]);
      (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
      process_buffer(buffer, cnt, dt_v);
      // Put the buffer's position and velocity data back into main storage
      spu_mfcdma32((void *)(&pos[buffer][0]), (unsigned int)(ctx_pos_v), cnt *
        sizeof(vector float), tags[buffer], MFC_PUT_CMD);
      spu_mfcdma32((void *)(&vel[buffer][0]), (unsigned int)(ctx_vel_v), cnt *
        sizeof(vector float), tags[buffer], MFC_PUT_CMD);

      ctx_pos_v = next_ctx_pos_v;
      ctx_vel_v = next_ctx_vel_v;
      ctx_inv_mass = next_ctx_inv_mass;
      buffer = next_buffer;
      cnt = next_cnt;
    }
    // Wait for previously prefetched data
    spu_writech(MFC_WrTagMask, 1 << tags[buffer]);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
    process_buffer(buffer, cnt, dt_v);
    // Put the buffer's position and velocity data back into main storage
    spu_mfcdma32((void *)(&pos[buffer][0]), (unsigned int)(ctx_pos_v), cnt *
      sizeof(vector float), tags[buffer], MFC_PUT_CMD);
    spu_mfcdma32((void *)(&vel[buffer][0]), (unsigned int)(ctx_vel_v), cnt *
      sizeof(vector float), tags[buffer], MFC_PUT_CMD);
    // Wait for DMAs to complete before starting the next step in time.
    spu_writech(MFC_WrTagMask, 1 << tags[buffer]);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  }
  return (0);
}
```

Static analysis of optimization

The listing below shows a *spu_timing* static timing analysis for the optimized SPE thread (process `_buffer` subroutine only).

```
.type   process_buffer, @function
                                                                    process_buffer:
0D 0123                                                              shli    $2,$3,10
1D 012345                                                            lqa     $19,ctx+16
0D  12                                                               ori     $6,$3,0
1D  1234                                                             shlqbyi $24,$4,0
0D   23                                                              cgti    $3,$4,0
1D   2345                                                            shlqbyi $18,$5,0
0D    34                                                             ila     $4,inv_mass
1D    3456                                                           fsmbi   $21,0
0      45                                                            ilhu    $27,1029
0       56                                                           ilhu    $26,2057
0        67                                                          ilhu    $25,3085
0         78                                                         ila     $28,66051
0          89                                                        a       $20,$2,$4
0           90                                                       iohl    $27,1543
0D           01                                                      iohl    $26,2571
1D           0                                                       lnop
0D            12                                                     iohl    $25,3599
1D            1234                                                   brz     $3,.L7
```

```
0              2345                                                  shli    $17,$6,14
0               34                                                   ila     $23,pos
0D               45                                                  ila     $22,vel
1D               456789                                              hbra    .L10,.L5
1                 5                                                  lnop
0                  6                                                 nop     $127
                                                                    .L5:
0D                  78                                              ila     $43,pos
1D                  789012                                          lqd     $41,0($20)
0D                   89                                             ila     $42,vel
1D                   890123                                         lqx     $40,$17,$23
0                     90                                            a       $6,$17,$43
0                      01                                           a       $7,$17,$42
0D                      12                                          ai      $21,$21,4
1D                      123456                                      lqd     $39,16($6)
0D                       23                                         ai      $20,$20,16
1D                       234567                                     lqd     $38,32($6)
0D                        345678                                    fm      $36,$18,$41
1D                        345678                                    lqd     $37,48($6)
0D                         45                                       cgt     $16,$24,$21
1D                         456789                                   lqx     $13,$17,$22
1                           567890                                  lqd     $34,16($7)
1                            678901                                 lqd     $14,32($7)
1                             789012                                lqd     $15,48($7)
1                              -9012                                shufb   $35,$36,$36,$28
0D                               012345                            fma     $32,$13,$18,$40
1D                               0123                              shufb   $33,$36,$36,$27
0D                                123456                           fma     $10,$34,$18,$39
1D                                1234                             shufb   $31,$36,$36,$26
0D                                 234567                          fma     $11,$14,$18,$38
1D                                 2345                            shufb   $30,$36,$36,$25
0                                   345678                         fma     $8,$15,$18,$37
0                                    456789                        fma     $29,$35,$19,$13
0D                                    567890                       fma     $5,$33,$19,$34
1D                                    5                            lnop
0D                                     678901                      fma     $12,$31,$19,$14
1D                                     678901                      stqx    $32,$17,$23
0D                                      789012                     fma     $9,$30,$19,$15
1D                                      789012                     stqd    $10,16($6)
1                                        890123                    stqd    $11,32($6)
1                                         901234                   stqd    $8,48($6)
0D                                         0                       nop     $127
1D                                         012345                  stqx    $29,$17,$22
0D                                          12                     ai      $17,$17,64
1D                                          123456                 stqd    $5,16($7)
1                                            234567                stqd    $12,32($7)
1                                             345678               stqd    $9,48($7)
0D                                             4                   nop     $127
                                                                    .L10:
1D                                             4567                brnz    $16,.L5
                                                                    .L7:
0D                                              5                  nop     $127
1D                                              5678               bi      $lr
```

Dynamic analysis of optimizations

The listing below shows a dynamic timing analysis on the *IBM Full System Simulator for the Cell Broadband Engine* for the optimized SPE thread (process buffer only). It shows that 78 registers are used, so the used percentage is 60.94.

```
SPU DD1.0
***
Total Cycle count               7134843
Total Instruction count         10602009
Total CPI                       0.67
***
Performance Cycle count         7134843
Performance Instruction count   10602009 (9839265)
Performance CPI                 0.67 (0.73)

Branch instructions             253940
Branch taken                    251967
Branch not taken                1973

Hint instructions               2952
Hint hit                        250980

Contention at LS between Load/Store and Prefetch 6871

Single cycle                                          3815689 ( 53.5%)
Dual cycle                                            3011788 ( 42.2%)
Nop cycle                                                5898 (  0.1%)
Stall due to branch miss                                34655 (  0.5%)
Stall due to prefetch miss                                  0 (  0.0%)
Stall due to dependency                                266732 (  3.7%)
Stall due to fp resource conflict                           0 (  0.0%)
Stall due to waiting for hint target                       72 (  0.0%)
Stall due to dp pipeline                                    0 (  0.0%)
Channel stall cycle                                         0 (  0.0%)
SPU Initialization cycle                                    9 (  0.0%)
-------------------------------------------------------------------------
Total cycle                                           7134843 (100.0%)

Stall cycles due to dependency on each pipelines
 FX2        8808
 SHUF       1971
 FX3        5870
 LS         32
 BR         0
 SPR        1
 LNOP       0
 NOP        0
 FXB        0
 FP6        250050
 FP7        0
 FPD        0
The number of used registers are 78, the used ratio is 60.94
```

The above static and dynamic timing analysis of the optimized SPE code reveals:

- Significant increase in dual-issue rate and reduction in dependency stalls. The static analysis shows that the `process_buffer` inner loop still contains a single-cycle stall and some instructions that are not dual-issued. Further performance improvements could likely be achieved by either more loop unrolling or software loop-pipelining.
- The number of instructions has decreased by 41% from the initial instruction count.
- The CPI has dropped from 2.39 to a more typical 0.73.
- The performance of the SPE code, measured in total cycle count, has gone from approximately 43 M cycles to 7 M cycles, an improvement of more than 6x. This improvement does not take into account the DMA latency-hiding (stall elimination) provided by double buffering.

General SPE programming tips

This section contains a short summary of general tips for optimizing the performance of SPE programs.

- *Local Store*
  - Design for the LS size. The LS holds up to 256 KB for the program, stack, local data structures, and DMA buffers. One can do a lot with 256 KB, but be aware of this size.
  - Use overlays (runtime download program kernels) to build complex function servers in the LS. This can be done using SPE overlays.
- *DMA Transfers*
  - Use SPE-initiated DMA transfers rather than PPE-initiated DMA transfers. There are more SPEs than the one PPE, and the PPE can enqueue only eight DMA requests whereas each SPE can enqueue 16.
  - Overlap DMA with computation by double buffering or multibuffering. Multibuffer code or (typically) data.
  - Use double buffering to hide memory latency.
  - Use `fence` command options to order DMA transfers within a tag group.
  - Use `barrier` command options to order DMA transfers within the queue.
- *Loops*
  - Unroll loops to reduce dependencies and increase dual-issue rates. This exploits the large SPU register file.
  - Compiler auto-unrolling is not perfect, but pretty good.
- *SIMD Strategy*
  - Choose an SIMD strategy appropriate for your algorithm. For example:
  - Evaluate array-of-structure (AOS) organization. For graphics vertices, this organization (also called or vector-across) can have more-efficient code size and simpler DMA needs, but less-efficient computation unless the code is unrolled.
  - Evaluate structure-of-arrays (SOA) organization. For graphics vertices, this organization (also called parallel-array) can be easier to `SIMDize`, but the data must be maintained in separate arrays or the SPU must shuffle AOS data into an SOA form.
  - Consider the effects of unrolling when choosing an SIMD strategy.
- *Load/Store*
  - Scalar loads and stores are slow, with long latency.
  - SPUs only support quadword loads and stores.
  - Consider making scalars into quadword integer vectors.
  - Load or store scalar arrays as quadwords, and perform your own extraction and insertion to eliminate load and store instructions.
- *Branches*
  - Eliminate nonpredicted branches.
  - Use feedback-directed optimization.
  - Use the `__builtin_expect` language directive when you can explicitly direct branch prediction.
- *Multiplies*
  - Avoid integer multiplies on operands greater than 16 bits in size. The SPU supports only a "16-bit x16-bit multiply". A "32-bit multiply" requires five instructions (three 16-bit multiplies and two adds).
  - Keep array elements sized to a power-of-2 to avoid multiplies when indexing.
  - Cast operands to `unsigned short` prior to multiplying. Constants are of type `int` and also require casting. Use a macro to explicitly perform 16-bit multiplies. This can avoid inadvertent introduction of signed extends and masks due to casting.
- *Pointers*
  - Use the PPE's `load/store with update` instructions. These allow sequential indexing through an array without the need of additional instructions to increment the array pointer.
  - For the SPEs (which do not support `load/store with update` instructions), use the d-form instructions to specify an immediate offset from a base array pointer.
- *Dual-Issue*
  - Choose intrinsics carefully to maximize dual-issue rates or reduce latencies.
  - Dual issue will occur if a `pipe-0` instruction is even-addressed, a `pipe-1` instruction is

odd-addressed, and there are no dependencies (operands are available).

- Code generators use `nops` to align instructions for dual-issue.
- Use software pipeline loops to improve dual-issue rates.