Start-up example: SDK 3.1 Matrix multiplication example.

Start-up example in order to familiarize using a Cell/B.E.-based application.

Main objectives of this example are:

1. Look and familiarize with SPE thread creation code.
2. Look and familiarize with application build steps.
3. Run different implementations of matrix multiplication and notice how optimizations affect performance.
4. Notice how memory placement affects performance.

1. SPE Thread creation code

First, take a look at thread creation lines in PPE code and SPE main subroutine.

PPE Code **(file matrix_mul/matrix_mul.c)**

```
    …
    …
    …

/* Per thread state
 */
struct _threads {
  spe_context_ptr_t id;
  pthread_t pthread;
  spe_spu_control_area_t *ctl_area;             // pointer to control ps area
  int in_cnt;                                   // inbound mailbox available element count
  mm_parms parms;
} threads[MAX_SPUS];

← Pthread that manages SPE Context →

static void *ppu_pthread_function(void *arg) {
  struct _threads *data;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  data = (struct _threads *)arg;

  if (spe_context_run(data->id, &entry, 0, (void *)(&(data->parms)), NULL, NULL) < 0) {
    perror("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}

extern spe_program_handle_t matrix_mul_spu;

    …
    …
    …

← Loop that allocates SPE Contexts, load the program in the SPE context and create the Pthread that will manage the SPE Context →

  /* Create each of the SPU threads
   */
  for (i=0; i<spus; i++) {
    /* Initialize the thread structure and its parameters.
     */
    threads[i].in_cnt = 0;
    threads[i].parms.id = i;

    …
    …
    …

    /* Create context */
    if ((threads[i].id = spe_context_create (SPE_MAP_PS, NULL)) == NULL) {
      printf("INTERNAL ERROR: failed to create spu context %d. Error = %s\n", i, strerror(errno));
      exit(1);
    }

    /* Load program */
    if ((rc = spe_program_load (threads[i].id, &matrix_mul_spu)) != 0) {
      printf("INTERNAL ERROR: failed to load program %d. Error = %s\n", i, strerror(errno));
      exit(1);
    }
    /* Create thread */
    if ((rc = pthread_create (&threads[i].pthread, NULL, &ppu_pthread_function, &threads[i].id)) != 0) {
```

```
      printf("INTERNAL ERROR: failed to create pthread %d. Error = %s\n", i, strerror(errno));
      exit(1);
    }
    if ((threads[i].ctl_area = (spe_spu_control_area_t *)spe_ps_area_get(threads[i].id, SPE_CONTROL_AREA)) == NULL) {
      printf("INTERNAL ERROR: failed to get control problem state area for thread %d. Error = %s\n", i, strerror(errno));
      exit(1);
    }
    threads[i].in_cnt = 0;
  }
```

SPE Code (file matrix_mul/spu/matrix_mul_spu.c)

```
int main(unsigned long long speid __attribute__ ((unused)), unsigned long long parms_ea)
{
  unsigned int i, j, k, iter, iters;
  unsigned int i_next, j_next, mail, tiles, msize, ld;
  unsigned int sync_count, spus;
  unsigned int in_idx = 0;
  unsigned int out_idx = 0;
    …
    …
    …
```

2. Build MatrixMult application

To build the application just type make. SPU Code part is built and embedded in a PPE library which is linked with SPE Code.

```
cd matrix_mul
make

← SPU code built with spu-gcc →
make[1]: Entering directory `/gpfs/data/home/Extern/mcuser00/winterschool/matrix_mul/spu'
/usr/bin/spu-gcc -W -Wall -Winline -I. -I.. -I /opt/cell/sdk/usr/spu/include -O3 -std=c99 -funroll-loops -O3 -c mat_mult_MxM_scalar.c
/usr/bin/spu-gcc -W -Wall -Winline -I. -I.. -I /opt/cell/sdk/usr/spu/include -O3 -std=c99 -funroll-loops -ftree-vectorize -O3 -c mat_mult_MxM_autovec.c
/usr/bin/spu-gcc -W -Wall -Winline -I. -I.. -I /opt/cell/sdk/usr/spu/include -O3 -std=c99 -funroll-loops -O3 -c mat_mult_MxM_simd.c
/usr/bin/spu-gcc -W -Wall -Winline -I. -I.. -I /opt/cell/sdk/usr/spu/include -O -std=c99 -funroll-loops -O3 -c mat_mult_64x64_opt_simd.c
/usr/bin/spu-gcc -E -W -Wall -Winline  -I. -I.. -I /opt/cell/sdk/usr/spu/include mat_mult_64x64_asm.S | /usr/bin/spu-as  -o mat_mult_64x64_asm.o
/usr/bin/spu-gcc W -Wall -Winline -I. -I.. -I /opt/cell/sdk/usr/spu/include -O3 -std=c99 -funroll-loops -O3 -c matrix_mul_spu.c


← Link everything in a SPE binary →
/usr/bin/spu-gcc -o matrix_mul_spu mat_mult_MxM_scalar.o mat_mult_MxM_autovec.o mat_mult_MxM_simd.o mat_mult_64x64_opt_simd.o mat_mult_64x64_asm.o
matrix_mul_spu.o -Wl,-N

← Embed SPE binary in a PPE object and assign a symbol to it →
/usr/bin/ppu-embedspu -m32 matrix_mul_spu matrix_mul_spu matrix_mul_spu-embed.o

← Include PPE object in a library →
/usr/bin/ppu-ar -qcs lib_matrix_mul_spu.a matrix_mul_spu-embed.o
make[1]: Leaving directory `/gpfs/data/home/Extern/mcuser00/winterschool/matrix_mul/spu'

← Build PPE code and link it with matrix_mul_spu.a library →
/usr/bin/ppu32-gcc -W -Wall -Winline -m32 -I. -I /opt/cell/sdk/usr/include -mabi=altivec -maltivec -O3 -c matrix_mul.c
/usr/bin/ppu32-gcc -o matrix_mul matrix_mul.o -L/opt/cell/sdk/usr/lib -R/opt/cell/sdk/usr/lib  -m32 -Wl,-m,elf32ppc spu/lib_matrix_mul_spu.a -lmisc -lspe2 -
lpthread -lm -lnuma
```

3. Run the different kernel implementations

Matrix multiplication example contains different implementations of basic 64x64 matrix multiplication kernel.

Objectives:

- Look the sources of the different kernel implementations. Notice the different levels of complexity: from simple scalar code to highly optimized assembler code.

- Run each implementation and annotate on the table the performance results. Notice the difference in performance in each implementation.

Simple scalar implementation

Source file: matrix_mul/spu/mat_mult_MxM_scalar.c
Command: ./matrix_mul -p -H -i 3 -m 4096 -s 8 **-o 0**

Simple scalar impl. & some auto-vectorization

Source file: matrix_mul/spu/mat_mult_MxM_autovec.c
Command: ./matrix_mul -p -H -i 3 -m 4096 -s 8 **-o 1**

Simple SIMD implementation

Source file: matrix_mul/spu/mat_mult_MxM_simd.c
Command: ./matrix_mul -p -H -i 3 -m 4096 -s 8 **-o 2**

Optimized SIMD implementation

Source file: matrix_mul/spu/mat_mult_64x64_opt_simd.c
Command: ./matrix_mul -p -H -i 3 -m 4096 -s 8 **-o 3**

Higly optimized SIMD implementation

Source file: matrix_mul/spu/ mat_mult_64x64_asm.S
Command: ./matrix_mul -p -H -i 3 -m 4096 -s 8 **-o 4**

Code version	Execution Time (sec)	Gflops/sec	GBytes/sec
Simple scalar			
Simple scalar compiled with autovectorization			
Simple SIMD			
Optimized SIMD			
Highly Optimized SIMD			

4. Control Memory Placement.

The IBM BladeCenter QS22 system is a NUMA machine. Each of both IBM PowerXcell 8i microprocessors have attached a set of DIMMs. Accesssing to chip-attached DIMMs is faster than accessing DIMMs attached to the other processor.

When an application is using more than one chip (for example, 16 SPUs), it is important where memory is placed. Numactl command allows

./matrix_mul -p -H -i 3 -m 4096 –o 4 **-s 16**

numactl --interleave=0,1 ./matrix_mul -p -H -i 3 -m 4096 –o 4 **-s 16**

	Execution Time	Gflops/sec	GBytes/sec
WITHOUT numactl			
WITH numactl			

Hands-on 1. Simple SPU creation and DMA sample.

Create a simple multi-threaded application that creates a number of SPU contexts, specified by argument, make each SPU context to fetch a structure from main memory using DMA (mfc_get) and prints its contents.

Structure to fetch (handson.h):

```
typedef struct _mm_parms {
  unsigned int id;           // SPU ID from 0 … #SPUs-1
  unsigned int spus;         // Total number of SPUs
  unsigned int pad[2];
} mm_parms;
```

We provide a **Skeleton** code to develop the example that can be found at *handson1/skeleton*.

1. Replace ***/* ADD*** comments with the corresponding call on the PPE code (look at previous matrix mult example)

```
  …
  …
  …
void *ppu_pthread_function(void *arg) {
  struct _threads *data;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  data = (struct _threads *)arg;

  /* ADD: start the SPU thread. Pass the structure address to the SPU thread as parameter. */

  pthread_exit(NULL);
}
  …
  …
  …
  /* Create several SPE-threads to execute 'handson_spu'. */
  for(i=0; i<spu_threads; i++) {

    threads[i].parms.id = i;
    threads[i].parms.spus = spu_threads;

    /* ADD: create  SPU context. Use thread struct fields to save the context. */

    /* ADD: load the handson_spu program */

    /* ADD: create the pthread that will run asynchronously to the main program.
       Make pthread to run "ppu_pthread_function" */
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<spu_threads; i++) {

    /* ADD: wait the pthread to finish */

    /* ADD: destroy the context */
  }
  …
  …
  …
```

2. Verify that code works. To build the code just type ‘make’

```
s1c1b2:/home/Extern/mcuser00/winterschool/handson1/solution2$./handson1 8

Example will create 8 SPU Threads

Hello from SPU (parms_ea 0x100175d0)
Hello from SPU (parms_ea 0x100175f0)
Hello from SPU (parms_ea 0x10017610)
Hello from SPU (parms_ea 0x10017630)
Hello from SPU (parms_ea 0x10017650)
Hello from SPU (parms_ea 0x10017670)
Hello from SPU (parms_ea 0x10017690)
Hello from SPU (parms_ea 0x100176b0)

The program has successfully executed.
```

3. Make SPU code to fetch the structure parms_ea and print its contents.

<mfc_get interface>

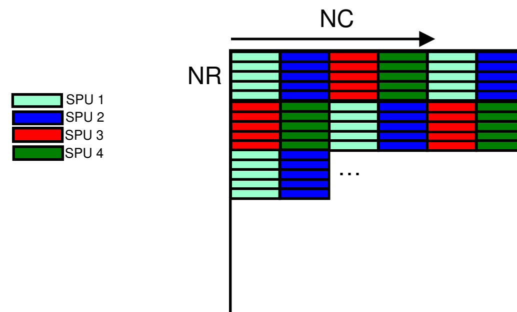
Solution code can be found at *handson1/solution*.

```
s1c1b2:/home/Extern/mcuser00/winterschool/handson1/solution$./handson1 8

Example will create 8 SPU Threads

Hello from SPU (parms_ea 0x100175d0)
parms_ea contents: id = 0 spus = 8
Hello from SPU (parms_ea 0x100175f0)
parms_ea contents: id = 1 spus = 8
Hello from SPU (parms_ea 0x10017610)
parms_ea contents: id = 2 spus = 8
Hello from SPU (parms_ea 0x10017630)
parms_ea contents: id = 3 spus = 8
Hello from SPU (parms_ea 0x10017650)
parms_ea contents: id = 4 spus = 8
Hello from SPU (parms_ea 0x10017670)
parms_ea contents: id = 5 spus = 8
Hello from SPU (parms_ea 0x10017690)
parms_ea contents: id = 6 spus = 8
Hello from SPU (parms_ea 0x100176b0)
parms_ea contents: id = 7 spus = 8

The program has successfully executed.
```

Hands-on 2. Matrix initialization.

Initialize the cells of a matrix on SPUs using the SPU ID.

The goal is to develop a code where SPUs fetch a matrix from main memory in blocks of 64x64 elements into SPU local memory, initialize the cells of the block using the value SPU ID+1 and send back that initialized block to their original location in main memory.



Considerations:

- Blocks are fetched in an interleaved way.

- Each block should be fetched using several DMAs. Memory organization requires to fetch several chucks of different rows to form a 64x64 block (Use of DMA list).

- SPU code must compute which blocks to fetch based on: matrix address, matrix size, SPU ID and number of SPUs. These data is passed using an structure similar to hasndon1 (handson.h).

```
typedef struct _mm_parms {
  unsigned int  id;
  unsigned int  spus;
  unsigned int  nra;
  unsigned int  nca;
  unsigned long long a;
  unsigned int  pad[2];
} mm_parms;
```

As previous example, we provide a **Skeleton** code that can be found at *handson2/skeleton*. Things to perform:

- PPE Code: Small replacement of `/* ADD` comment with the corresponding code.

- SPE Code: Replace `/* ADD` comments with the code that loops fetching the blocks, initializing them and put them back to main memory. Recommendation: use a DMA list.

DMA List

A **DMA list** is a sequence of transfer elements (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single area of LS and possibly discontinuous areas in main storage.

Such DMA lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command, such as getl or putl. Each transfer element in the DMA list contains a transfer

size, the low half of an effective address, and a stall-and-notify bit that can be used to suspend list execution after transferring a list element whose stall-and-notify bit is set.

```
#include <spu_mfcio.h>

  mfc_list_element_t dma_list[NUM_LIST_ELEMENTS] __attribute__ ((aligned (16)));

  for (ii = 0; ii < #elems_list; ii++)
  {
    dma_list[ii].notify = 0;
    dma_list [ii].size = <size>;
    dma_list [ii].eal = <address to fetch>;
  }


  mfc_getl (Blk, (unsigned long long) ini_addr, dma_list, #elems_list * sizeof(mfc_list_element_t), tag, 0, 0);
  mfc_putl (Blk, (unsigned long long) ini_addr, dma_list, #elems_list * sizeof(mfc_list_element_t), tag, 0, 0);
```

Solution code can be found at *handson2/solution*.

```
s1c1b2:/home/Extern/mcuser00/winterschool/handson1/solution$./handson1 8

Example will create 8 SPU Threads

Hello from SPU (parms_ea 0x100175d0)
parms_ea contents: id = 0 spus = 8
Hello from SPU (parms_ea 0x100175f0)
parms_ea contents: id = 1 spus = 8
Hello from SPU (parms_ea 0x10017610)
parms_ea contents: id = 2 spus = 8
Hello from SPU (parms_ea 0x10017630)
parms_ea contents: id = 3 spus = 8
Hello from SPU (parms_ea 0x10017650)
parms_ea contents: id = 4 spus = 8
Hello from SPU (parms_ea 0x10017670)
parms_ea contents: id = 5 spus = 8
Hello from SPU (parms_ea 0x10017690)
parms_ea contents: id = 6 spus = 8
Hello from SPU (parms_ea 0x100176b0)
parms_ea contents: id = 7 spus = 8

The program has successfully executed.
```

Hands-on 3. MPI Matrix addition.

Port a simple MPI Matrix addition code to the Cell/B.E applying the previous two hands-on.

The objective is to develop a code where SPUs fetch a matrix from main memory in blocks of 64x64 elements into SPU local memory, initialize the cells of the block using the value SPU ID+1 and copy back the initialized block to their original location in main memory.

We provide a simple MPI matrix add application where master MPI process distributes the matrix rows in several worker MPI processes that perform the adding of each part.

Base code is located at: ***mpi/matrixadd***.

Part I. Apply Hands-on1 exercise to include the creation of SPE Contexts that fetch the params struct with the matrix information. In this part, still keep the internal matrix addition loop as is.

Solution for Part I is at: ***mpi/matrixadd2***

Part II. Apply Hands-on2 to the MPI worker part to distribute the matrix addition loop between SPE contexts. Each SPE must fetch a block of A and B, add them, and put the result back to main memory. Use interleaved block distribution between SPEs.

Solution for Part II is at: ***mpi/matrixadd3***