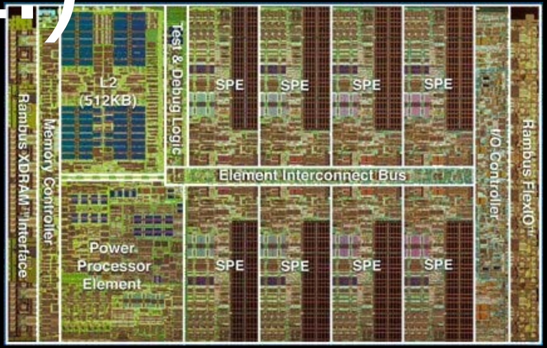
The Best Programming Practice for Cell/B.E.

2009.12.11

Sony Computer Entertainment Inc.
Akira Tsukamoto

Who am I?

Cell Broadband Engine (Cell/B.E.)

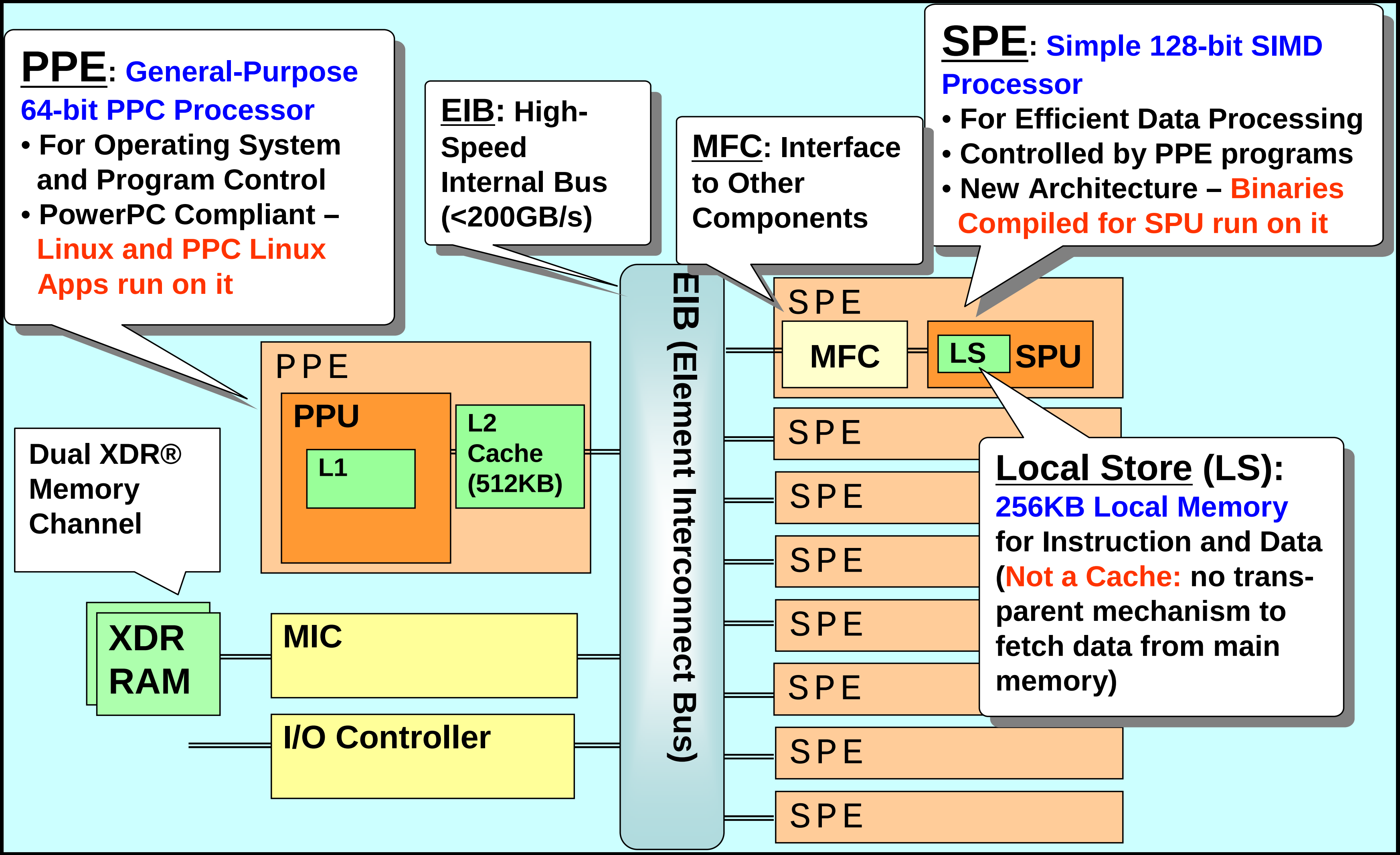


- 1 PPE: general-purpose
  - PowerPC architecture
  - System management
- 8 SPEs: between general-purpose and special-purpose
  - SIMD capable RISC architecture
  - 128 bit (16 byte) register size
  - 128 registers
  - Programs and data must be located on 256KiB local storage
  - External data access by DMA via MFC
  - Workloads for game, media, etc

Performance of Cell/B.E. Processor

- 9 Cores (1 PPE + 8 SPE)
- Peak Performance
  - Over 200 GFlops (Single Precision)
    - 4 (32-bit SIMD) * 2 (FMA) Flop
      * 8 SPE * 3.2GHz
      = 204.8 GFlops per socket
  - Over 20 GFlops (Double Precision)
  - Up to 25.6 GB/s Memory B/W
  - 35 GB/s (out) + 25 GB/s (in) I/O B/W
  - Over 200 GB/s Total Interconnect B/W
    - 96B/cycle

Overview of Cell/B.E.



PPE: General-Purpose 64-bit PPC Processor
- For Operating System and Program Control
- PowerPC Compliant – Linux and PPC Linux Apps run on it

EIB: High-Speed Internal Bus (<200GB/s)

MFC: Interface to Other Components

SPE: Simple 128-bit SIMD Processor
- For Efficient Data Processing
- Controlled by PPE programs
- New Architecture – Binaries Compiled for SPU run on it

Dual XDR® Memory Channel

Local Store (LS): 256KB Local Memory for Instruction and Data (Not a Cache: no trans-parent mechanism to fetch data from main memory)

The Future of Cell/B.E.

- PowerXCell32i (Quad Cell/B.E.) was canceled

BUT

- Current Cell/B.E. production will continue
  - PlayStation®3 (PS3®), IBM QS22 (CellBlade)
- SPE architecture will be incorporated to Power CPU

Cell/B.E. Programming skills are beneficial to achieve good performance on the future computer architecture

One big misunderstanding about Cell/B.E.

- Is SPE one kind of Floating Point Unit (FPU) or Digital Signaling Processor (DSP)?

NO

- SPE is one kind of regular CPU core but equipped with optimized Single Instruction Multiple Data (SIMD) operations.
- Could run program and process data in their memory called Local Storage (LS) as normal CPU does.

SPE vs. GPGPU

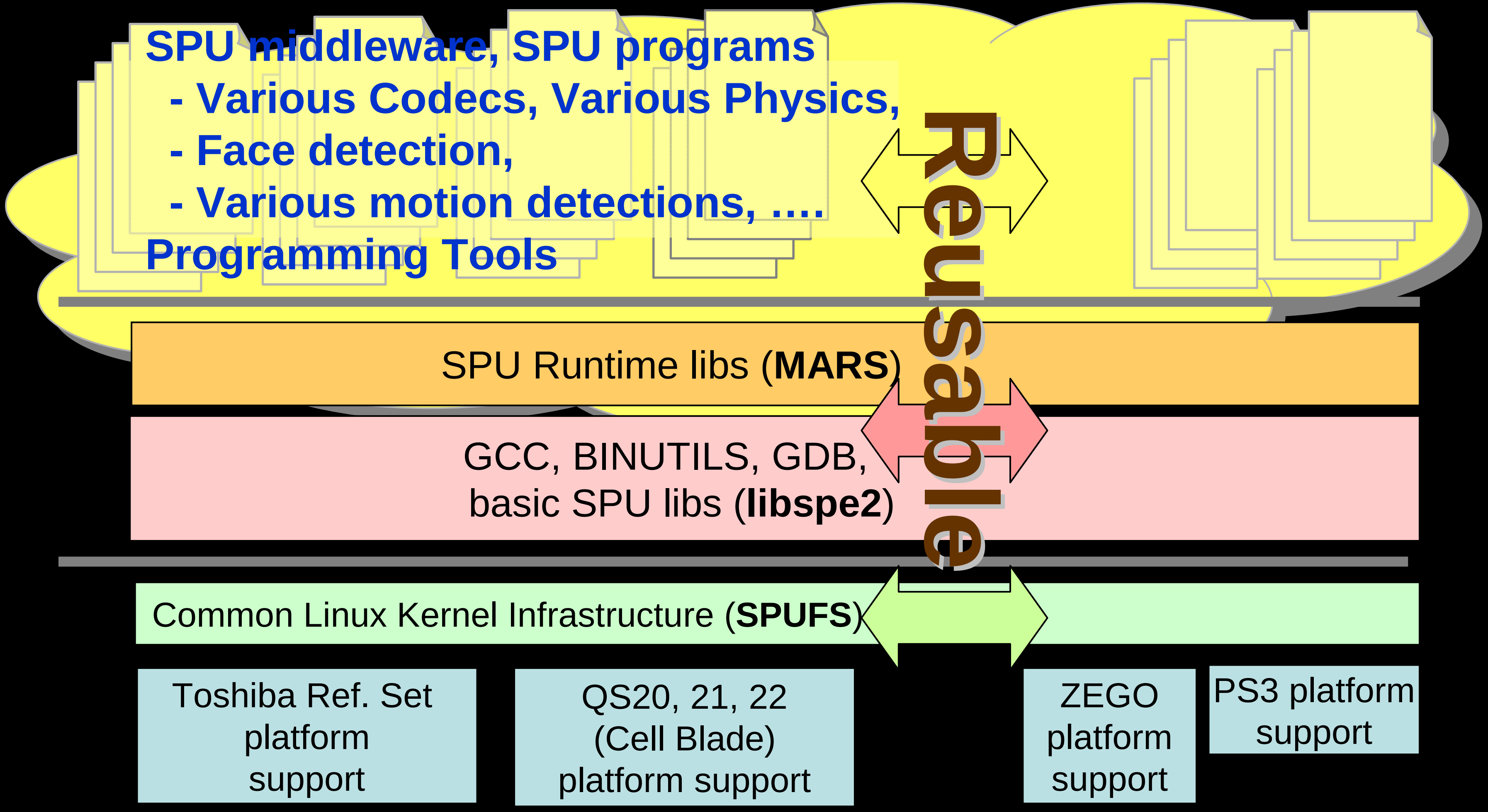
- SPE
  - Very good performance of general instructions
    - if(), switch(), for(), while() are fast in C/C++ language
  - Capable for different processing in parallel (Task parallel model)
    - 2 SPEs for Physics engine, 2 SPEs for vision recognition, 2 SPEs for codec
- GPGPU
  - Limited performance on general instructions
  - Not good for different processing in parallel (Task parallel model)
    - Suitable for processing large data with the same calculation (Data parallel model)

- SPE is better for general purpose processing to adopt wide range of programming

Cell/B.E. Programming Environment

- PPE toolchain
  - One of PowerPC targets
  - gcc and binutils with Cell/B.E. specific enhancements
- SPE toolchain
  - New target architecture
  - spu-gcc, binutils, newlib (libc), ...
- libspe
  - SPE management library
  - Provides OS-independent API by wrapping the raw SPUFS interfaces
- MARS
  - Provides effective SPE runtime environment

Cell/B.E. Programming Environment



SPU middleware, SPU programs
- Various Codecs, Various Physics,
- Face detection,
- Various motion detections, ….

Programming Tools

Reusable

SPU Runtime libs (**MARS**)

GCC, BINUTILS, GDB,
basic SPU libs (**libspe2**)

Common Linux Kernel Infrastructure (**SPUFS**)

Toshiba Ref. Set platform support

QS20, 21, 22 (Cell Blade) platform support

ZEGO platform support

PS3 platform support

Hello World Programming on Cell/B.E.

SPE programming

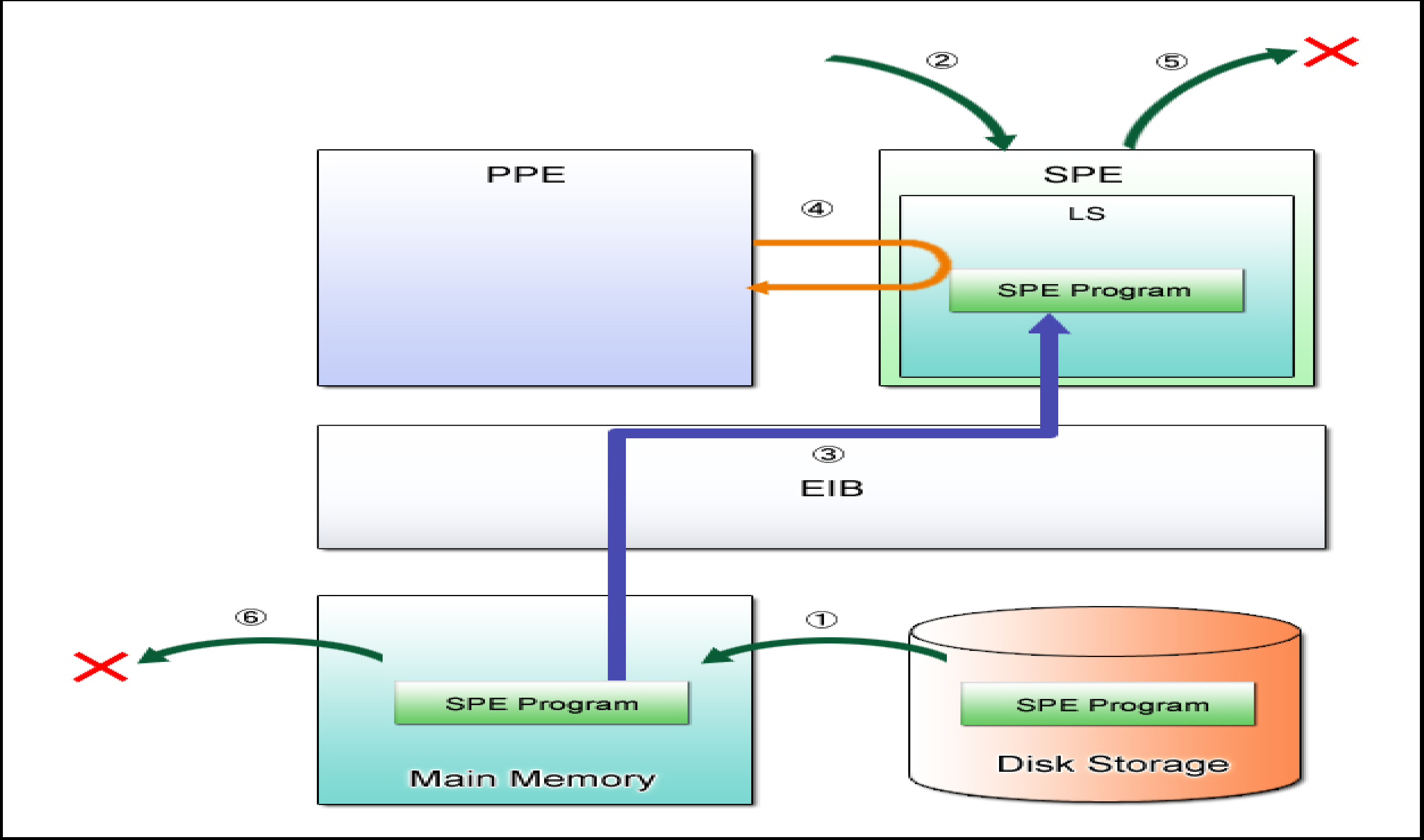
- SPE program prints “Hello World!”

```
#include <stdio.h>
int main()
{
    printf(“Hello, World!¥n”);
    return 0;
}
```

- SPE program prints “Hello World!”

```
$ spu-gcc hello.c -o hello.spe
$ ./hello.spe
```

Program execution flow on SPE



Optimizing SPE program

- Regular programming on SPE do not achieve Over 200 GFlops performance

- Requires optimization on SPE programming

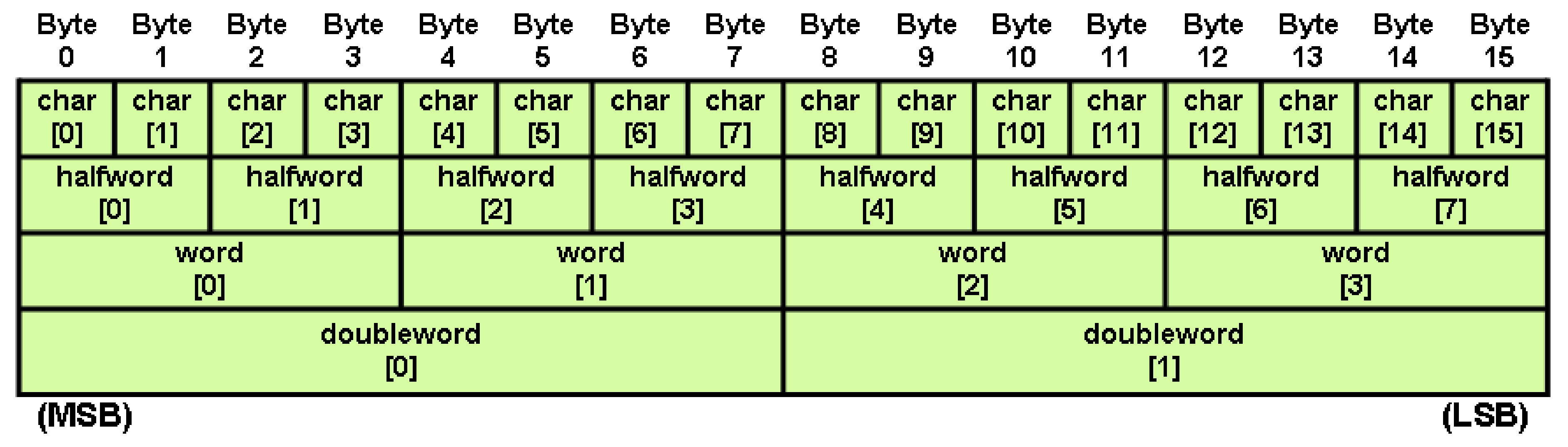
Optimization Technique on Cell/B.E.

Use SIMD Instructions

vector type extension on spu-gcc

Vector Type	Data
_vector unsigned char	Sixteen unsigned 8-bit data
_vector signed char	Sixteen signed 8-bit data
_vector unsigned short	Eight unsigned 16-bit data
_vector signed short	Eight signed 16-bit data
_vector unsigned int	Four unsigned 32-bit data
_vector signed int	Four signed 32-bit data
_vector unsigned long long	Two unsigned 64-bit data
_vector signed long long	Two signed 64-bit data
_vector float	Four single-precision floating-point data
_vector double	Two double-precision floating-point data

vector type extension on spu-gcc



Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15
char [0]	char [1]	char [2]	char [3]	char [4]	char [5]	char [6]	char [7]	char [8]	char [9]	char [10]	char [11]	char [12]	char [13]	char [14]	char [15]
halfword [0]		halfword [1]		halfword [2]		halfword [3]		halfword [4]		halfword [5]		halfword [6]		halfword [7]	
word [0]				word [1]				word [2]				word [3]			
doubleword [0]								doubleword [1]							

(MSB) (LSB)

SIMD programming

```
float a[4], b[4], c[4];

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

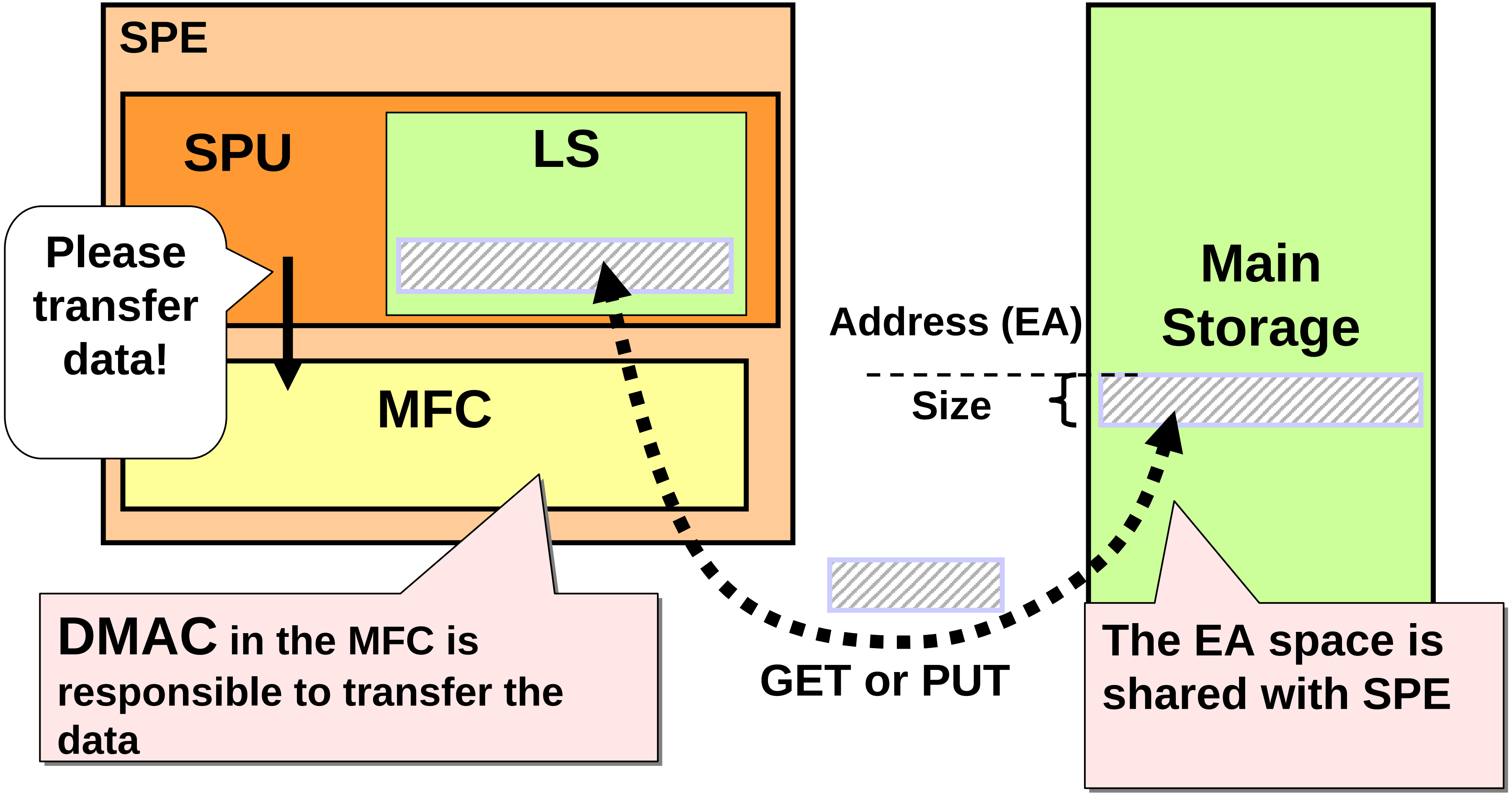
```
__vector float va, vb, vc;

vc = spu_mul(va, vb);
```

Other SIMD Built-in Functions

Applicable Instructions	VMX	SPU SIMD	Description
Arithmetic Instructions	vec_add(a,b)	spu_add(a,b)	Adds the elements of vectors *a* and *b.*
	vec_sub(a,b)	spu _sub(a,b)	Performs subtractions between the elements of vectors *a* and *b.*
	vec_madd(a,b,c)	spu_madd(a,b,c)	Multiplies the elements of vector *a* by the elements of vector *b* and adds the elements of vector *c.*
	vec_re(a,b)	spu_re(a,b)	Calculates the reciprocals of the elements of vector a.
	vec_rsqrte(a)	spu_rsqrte(a)	Calculates the square roots of the reciprocals of the elements of vector *a*
Logical Instructions	vec_and(a,b)	spu_and(a,b)	Finds the bitwise logical products (AND) between vectors *a* and *b*.
	vec_or(a,b)	spu_or(a,b)	Finds the bitwise logical sums (OR) between vectors *a* and *b*.

Use Double Buffering DMA between Main memory and LS

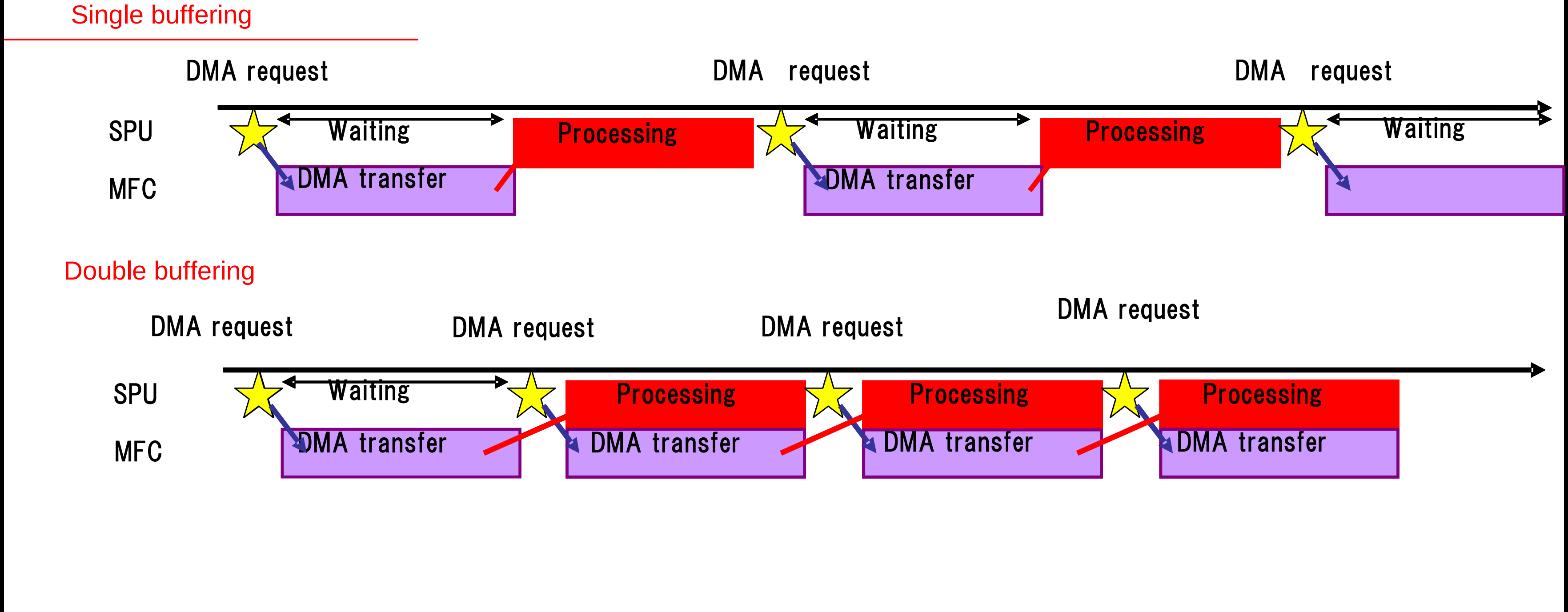
DMA between Main memory and LS



DMAC in the MFC is responsible to transfer the data

The EA space is shared with SPE

Single buffering and double buffering



Use Aligned Data

How to Align Data

- 128 byte aligned data is best for DMA
- 16 byte aligned data is best for SPE instructions
- Use gcc's aligned attribute for static or global data

```
__attribute__((aligned(align_size)))
```

  - Example: 16-bytes aligned integer variable

```
int a __attribute__((aligned(16)));
```

  - Example: defining a 128-bytes-aligned structure type

```
typedef struct { int a; char b; }
   __attribute__((aligned(128))) aligned_struct_t;
```

- Use posix_memalign for dynamic allocation

```
#define _XOPEN_SOURCE 600  /* include POSIX 6th definition */
#include <stdlib.h>
int posix_memalign(void **ptr, size_t 16, size_t size);
```

Use Loop Unrolling

Unroll for loop

- SPE has 128 entries of registers

```
for (i = 0; i < N; i += 4) {
  *(vec_float4*)&c[i] = *(vec_float4*)&a[i] *
                        *(vec_float4*)&b[i]);
}
```

```
for (i = 0; i < N; i += 16) {
  vec_float4 av0 = *(vec_float4*)(a + i);
  vec_float4 bv0 = *(vec_float4*)(b + i);
  vec_float4 av1 = *(vec_float4*)(a + i + 4);
  ...

  vec_float4 cv0 = av0 * bv0;
  vec_float4 cv1 = av1 * bv1;
  ...

  *(vec_float4*)(c + i) = cv0;
  *(vec_float4*)(c + i + 4) = cv1;
  ...
}
```

Load the input to registers

Compute on registers

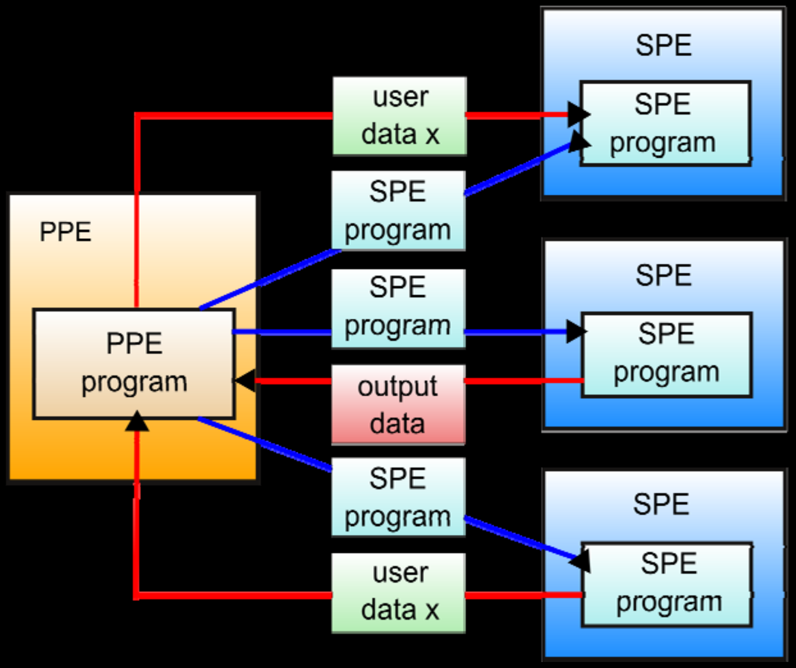
Effective Programming model of Cell/B.E.

Typical Cell/B.E. Program

- 1 PPE program
  - User interface
  - Data input/output
  - Loading and executing SPE programs
- Multiple SPE programs
  - Image processing
  - Physics simulation
  - Scientific calculation

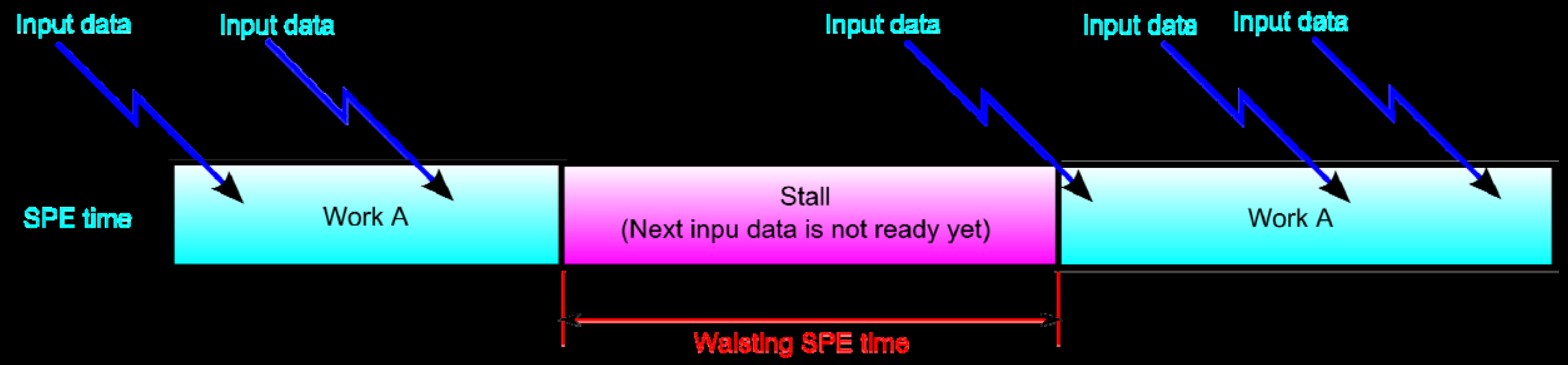
PPE Centric Programming Model

- PPE is responsible for:
  - Loading/switching of SPE programs
  - Sending/receiving of necessary data to its SPE programs



Problems of PPE Centric Programming

- Difficult for the PPE to know SPE's status
  - Stalls, waits...
  - Inefficient scheduling of SPE programs



- Extra load of the PPE
  - Communication
  - Scheduling

Preparation for MARS

Multi-core Application Runtime System

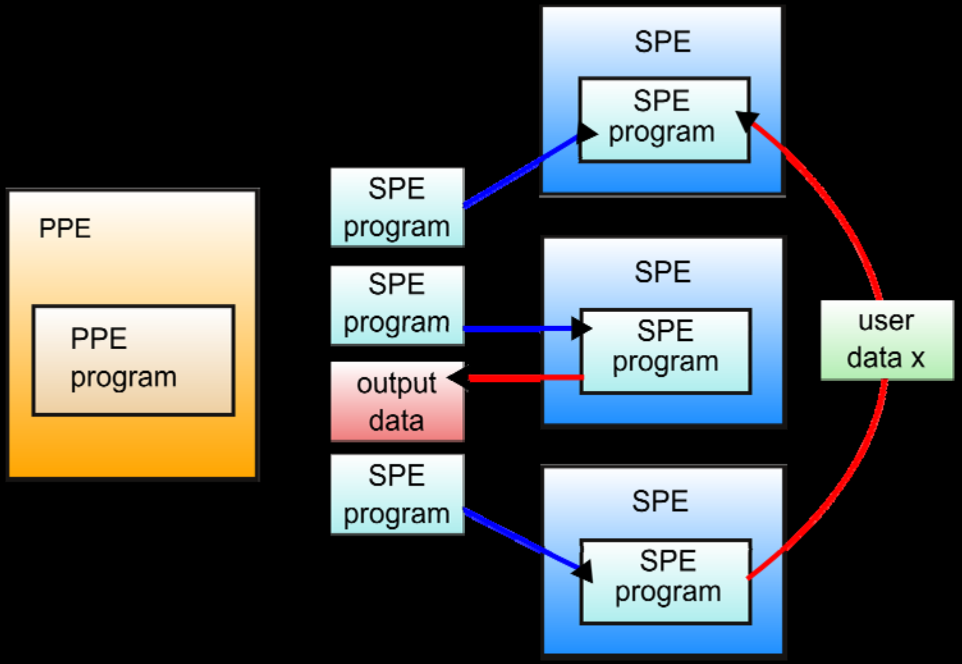
What is MARS?

MARS

- MARS stands for Multi-core Application Runtime System
- Provides efficient runtime environment for SPE centric application programs

SPE Centric Programming Model

- The individual SPEs are responsible for:
  - Loading, executing and switching SPE programs
  - Sending/receiving data between SPEs



What MARS Provides

- PPE Centric Programming model is slow
- Use PPE as less as possible

↓

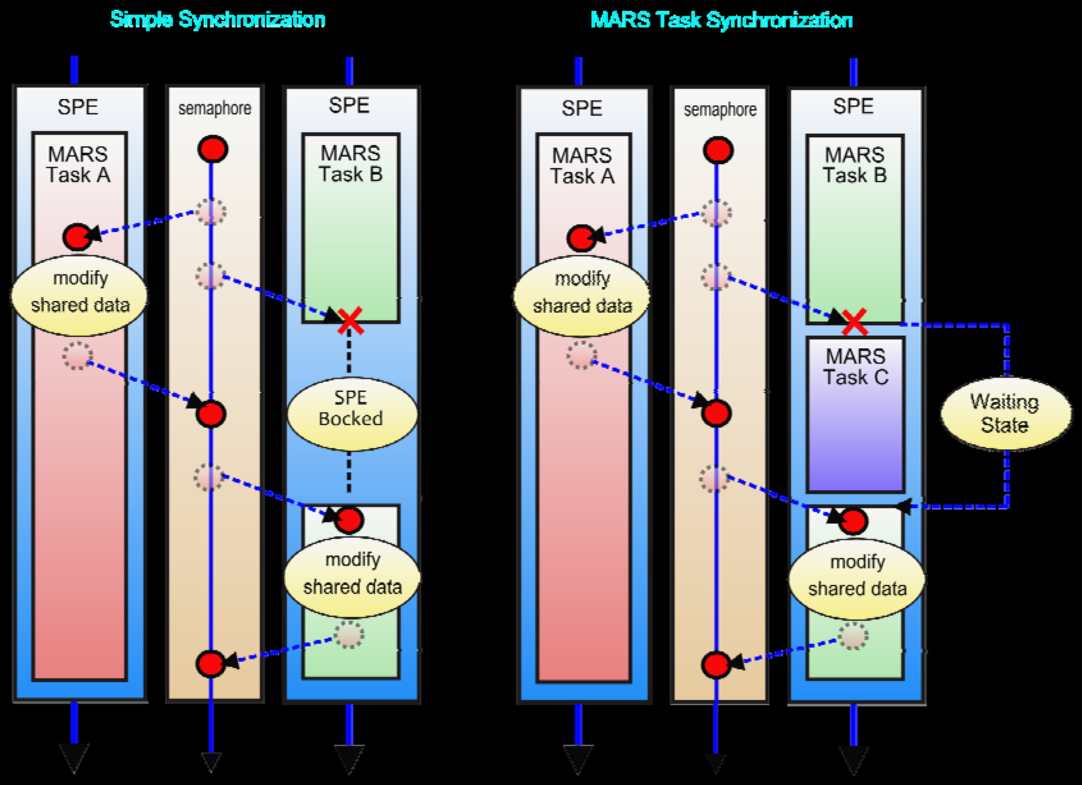
- MARS provides SPE centric runtime without complicate programming:
  - Scheduling workloads by SPEs
  - Lightweight context switching
  - Synchronization objects cooperating with the scheduler

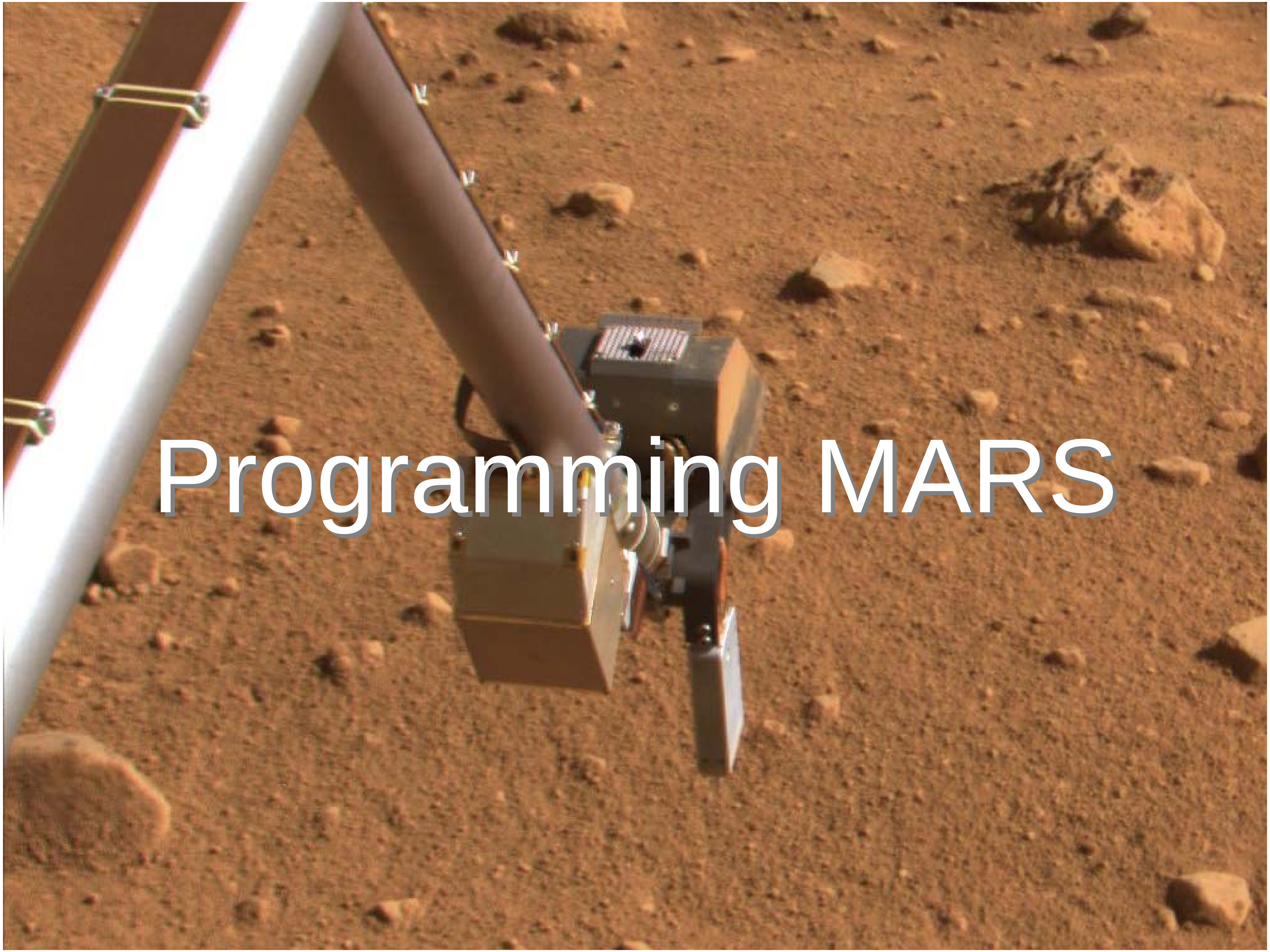
MARS Advantages

- Simplifies maximizing SPE usage
  - Efficient context switching
  - Minimizes data exchanged with PPE
- Minimizes runtime load of the PPE

MARS Task Sync Objects

- Semaphores, event flags, queues...
- Waiting condition results in a task switch
  - Avoiding wasting time just on waiting





Programming MARS

Typical Usage Scenario

1. PPE creates MARS context
2. PPE creates task objects
3. PPE creates synchronization objects
4. PPE starts the initial tasks
5. The existing tasks start additional tasks
6. The tasks do application specific works
7. PPE waits for tasks
8. PPE destroys task objects and sync objects
9. PPE destroys MARS context

Preparation program on PPE

```
int main(void)
{
        struct mars_context *mars_ctx;
        struct mars_task_id task1_id;
        static struct mars_task_id task2_id[NUM_SUB_TASKS] __attribute__((aligned(16)));
        struct mars_task_args task_args;
        int i;

        mars_context_create(&mars_ctx, 0, 0);

        mars_task_create(mars_ctx, &task1_id, "Task 1", spe_main_prog.elf_image,
                        MARS_TASK_CONTEXT_SAVE_ALL);

        for (i = 0; i < NUM_SUB_TASKS; i++) {
                char name[16];
                sprintf(name, "Task 2.%d", i);
                mars_task_create(mars_ctx, &task2_id[i], name, spe_calc_prog.elf_image,
                                MARS_TASK_CONTEXT_SAVE_ALL);
        }

        task_args.type.u64[0] = mars_ptr_to_ea(&task2_id[0]);
        task_args.type.u64[1] = mars_ptr_to_ea(&task2_id[1]);

        /* start main SPE MARS task */
        mars_task_schedule(&task1_id, &task_args, 0);

        mars_task_wait(&task1_id, NULL);
        mars_task_destroy(&task1_id);

        for (i = 0; i < NUM_SUB_TASKS; i++)
                mars_task_destroy(&task2_id[i]);

        mars_context_destroy(mars_ctx);

        return 0;
}
```

Main MARS task program on SPE

```c
#include <stdlib.h>
#include <spu_mfcio.h>
#include <mars/mars.h>

#define DMA_TAG 0

int mars_task_main(const struct mars_task_args *task_args)
{
        static struct mars_task_id task2_0_id __attribute__((aligned(16)));
        static struct mars_task_id task2_1_id __attribute__((aligned(16)));
        struct mars_task_args args;

        mfc_get(&task2_0_id, task_args->type.u64[0], sizeof(task2_0_id), DMA_TAG, 0, 0);
        mfc_get(&task2_1_id, task_args->type.u64[1], sizeof(task2_1_id), DMA_TAG, 0, 0);
        mfc_write_tag_mask(1 << DMA_TAG);
        mfc_read_tag_status_all();

        /* start calculation SPE MARS task 0 */
        args.type.u32[0] = 123;
        mars_task_schedule(&task2_0_id, &args, 0);

        /* start calculation SPE MARS task 1 */
        args.type.u32[0] = 321;
        mars_task_schedule(&task2_1_id, &args, 0);

        mars_task_wait(&task2_0_id, NULL);
        mars_task_wait(&task2_1_id, NULL);

        return 0;
}
```

Program for processing on SPE

```
#include <stdio.h>
#include <mars/mars.h>

int mars_task_main(const struct mars_task_args *task_args)
{

        /* do some calculations here */

        printf("MPU(%d): %s - Hello! (%d)¥n",
                mars_task_get_kernel_id(), mars_task_get_name(),
                task_args->type.u32[0]);

        return 0;
}
```

MARS synchronization API

- **Barrier**

  This is used to make multiple MARS tasks wait at a certain point in a program and to resume the task execution when all tasks are ready.

- **Event Flag**

  This is used to send event notifications between MARS tasks or between MARS tasks and host programs.

- **Queue**

  This is used to provide a FIFO queue mechanism for data transfer between MARS tasks or between MARS tasks and host programs.

- **Semaphore**

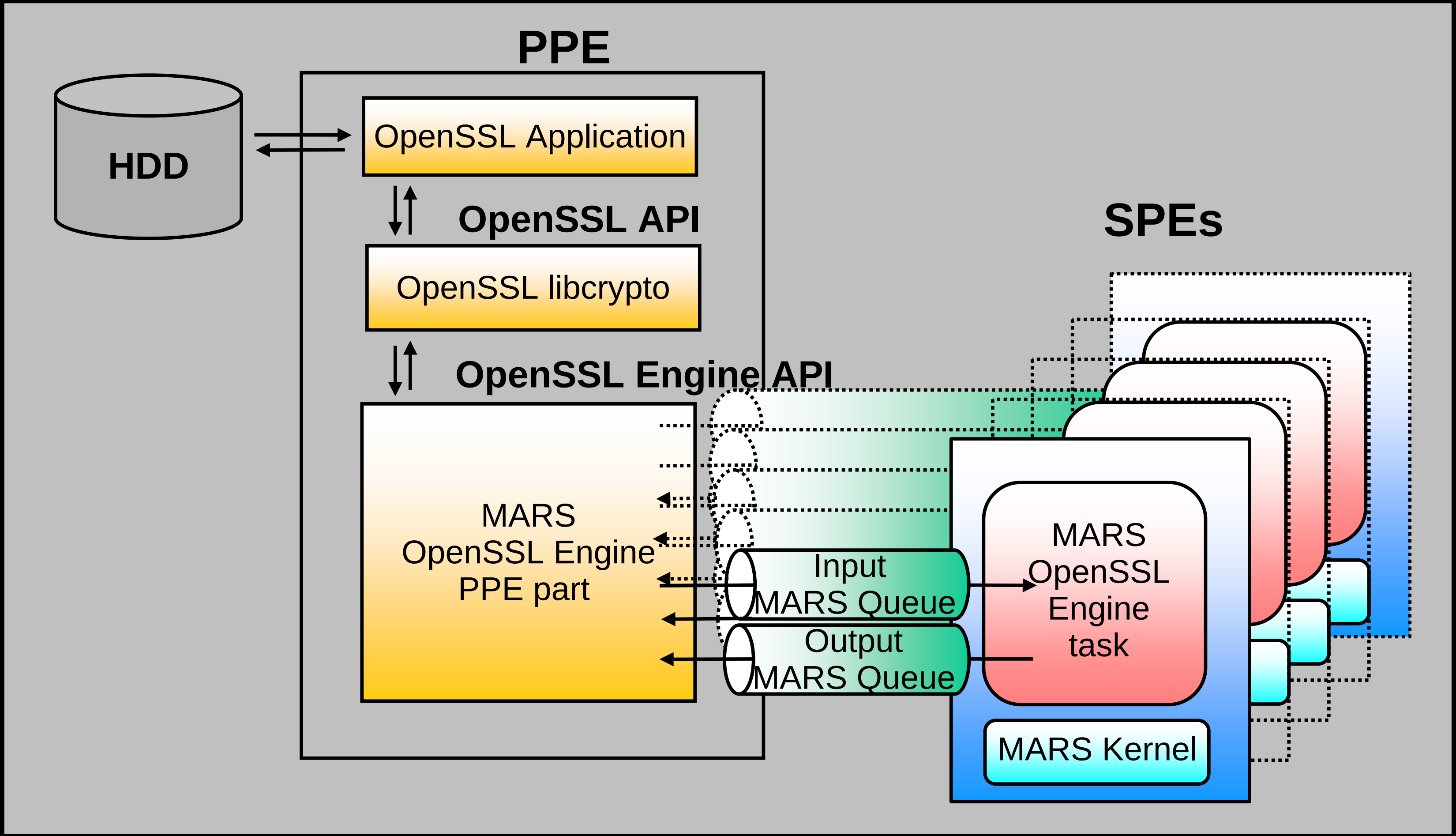
  This is used to limit the number of concurrent accesses to shared resources among MARS tasks.

- **Task Signal**

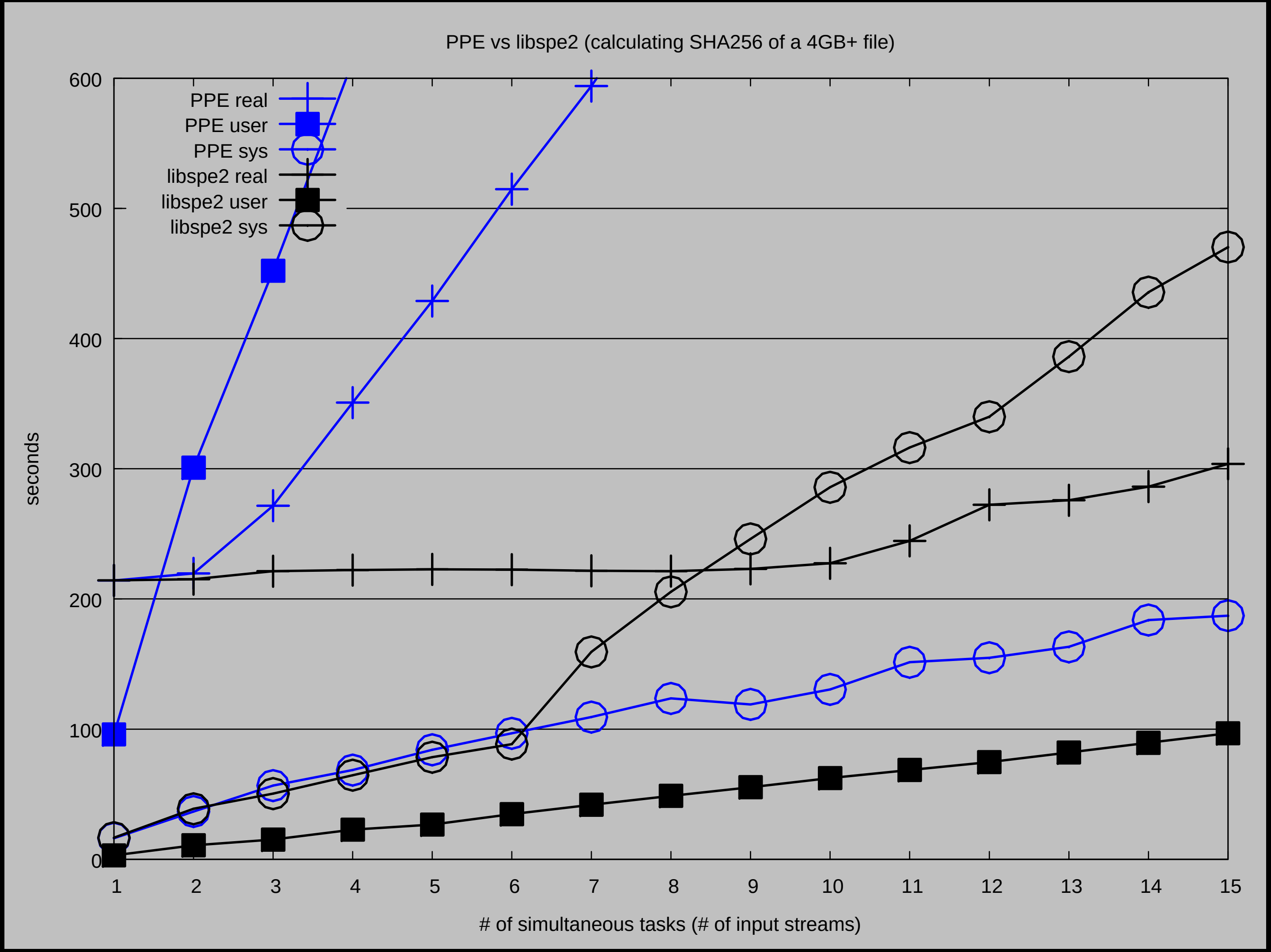
  This is used to signal a MARS task in the waiting state to change state so that it can be scheduled to continue execution.

Benchmark on MARS

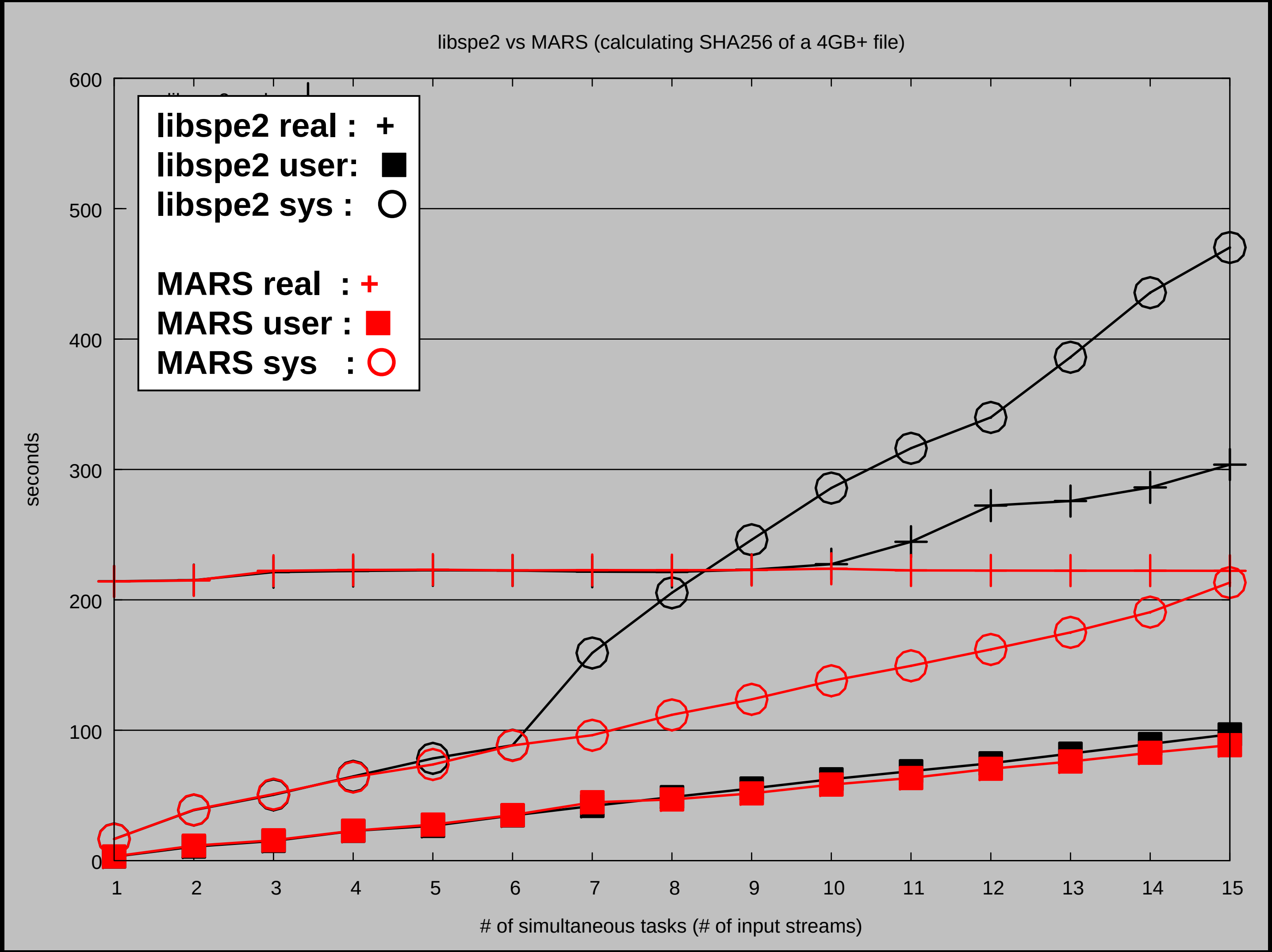
Sample Application: OpenSSL



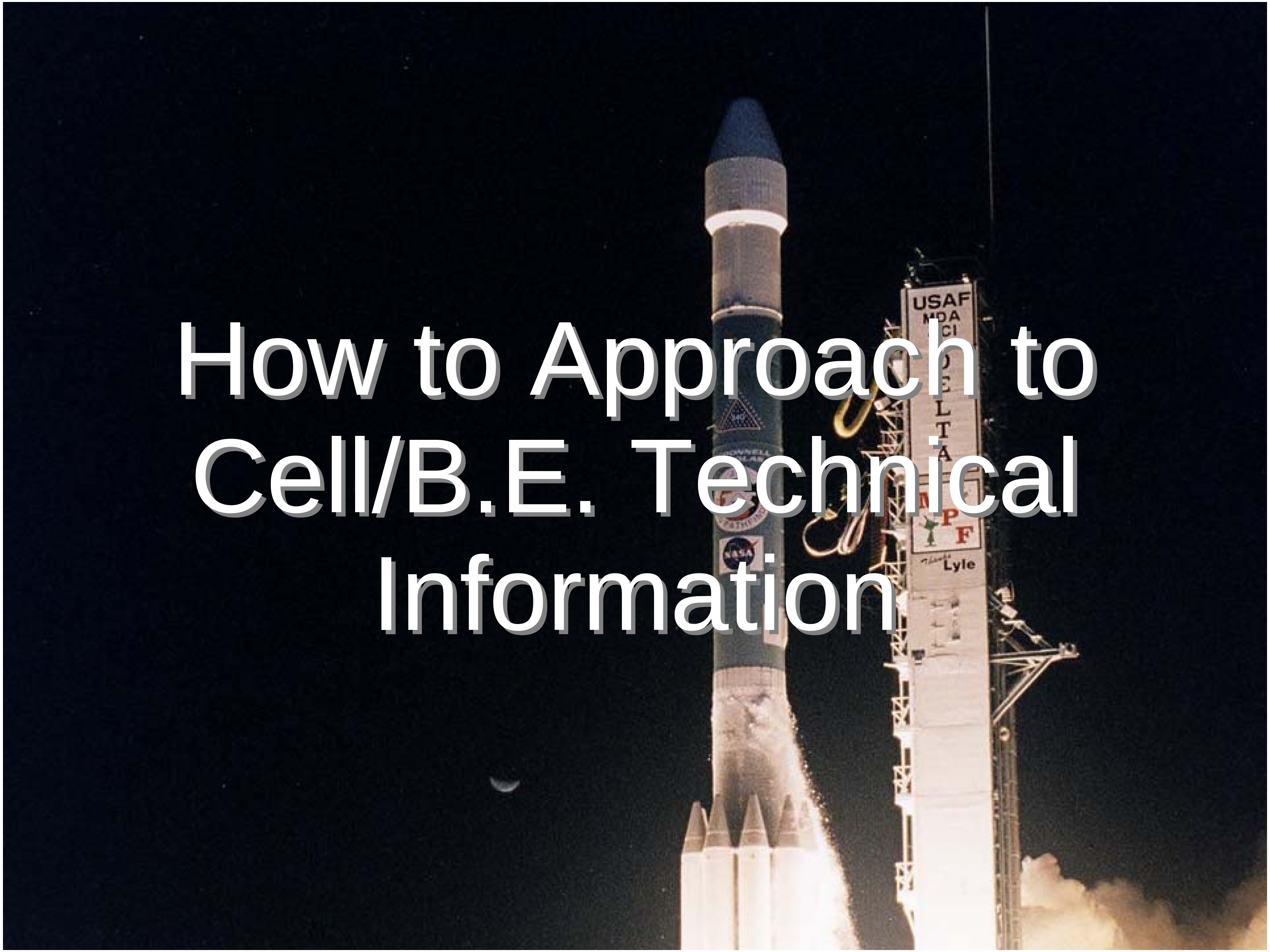
Benchmarking: OpenSSL
PPE vs SPE



Benchmarking: OpenSSL
libspe2 vs MARS



How to Approach to Cell/B.E. Technical Information



Information on Cell/B.E. programming

- **Cell/B.E. programming document**
  - http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingPrimer.html
- **PS3 Linux Public Information**
  - http://www.playstation.com/ps3-openplatform/index.html
  - http://www.kernel.org/pub/linux/kernel/people/geoff/cell/
- **Cell/B.E. information by IBM**
  - http://www.ibm.com/developerworks/power/cell/documents.html
  - http://www.bsc.es/projects/deepcomputing/linuxoncell/
- **Cell/B.E. Discussion Mailing List:**
  - cbe-oss-dev@ozlabs.org
  - https://ozlabs.org/mailman/listinfo/cbe-oss-dev
- **Cell/B.E. Discussion IRC:**
  - #cell at irc.freenode.org
- **MARS Releases, Source Code, Samples**
  - http://ftp.uk.linux.org/pub/linux/Sony-PS3/mars/
- **MARS Development Repositories:**
  - git://git.infradead.org/ps3/mars-src.git
  - http://git.infradead.org/ps3/mars-src.git