Обзор методов программирования гибридных вычислительных систем с графическими ускорителями

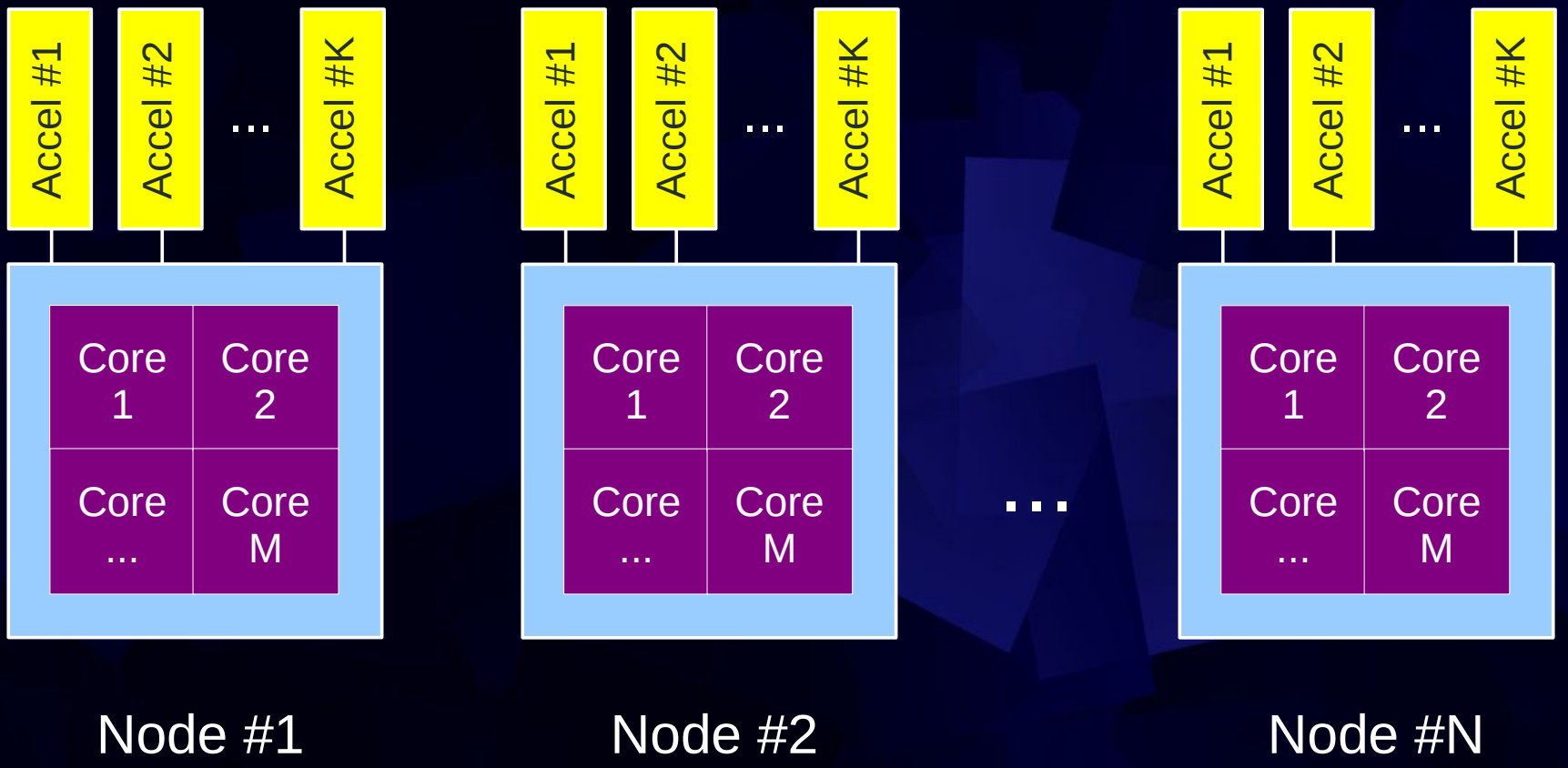
Микушин Д.Н.

maemarcus@gmail.com

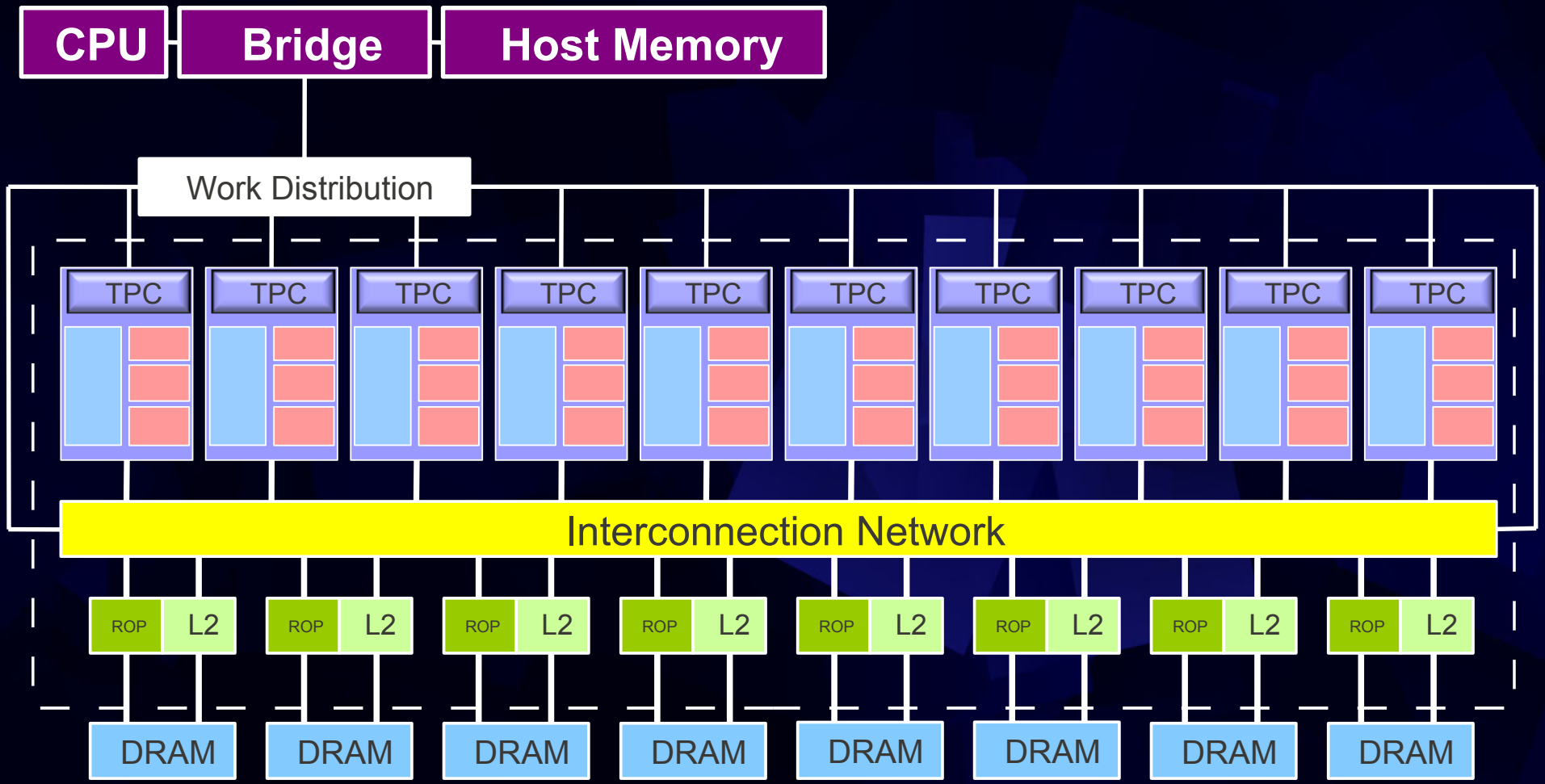
27 апреля 2010 г.

Содержание

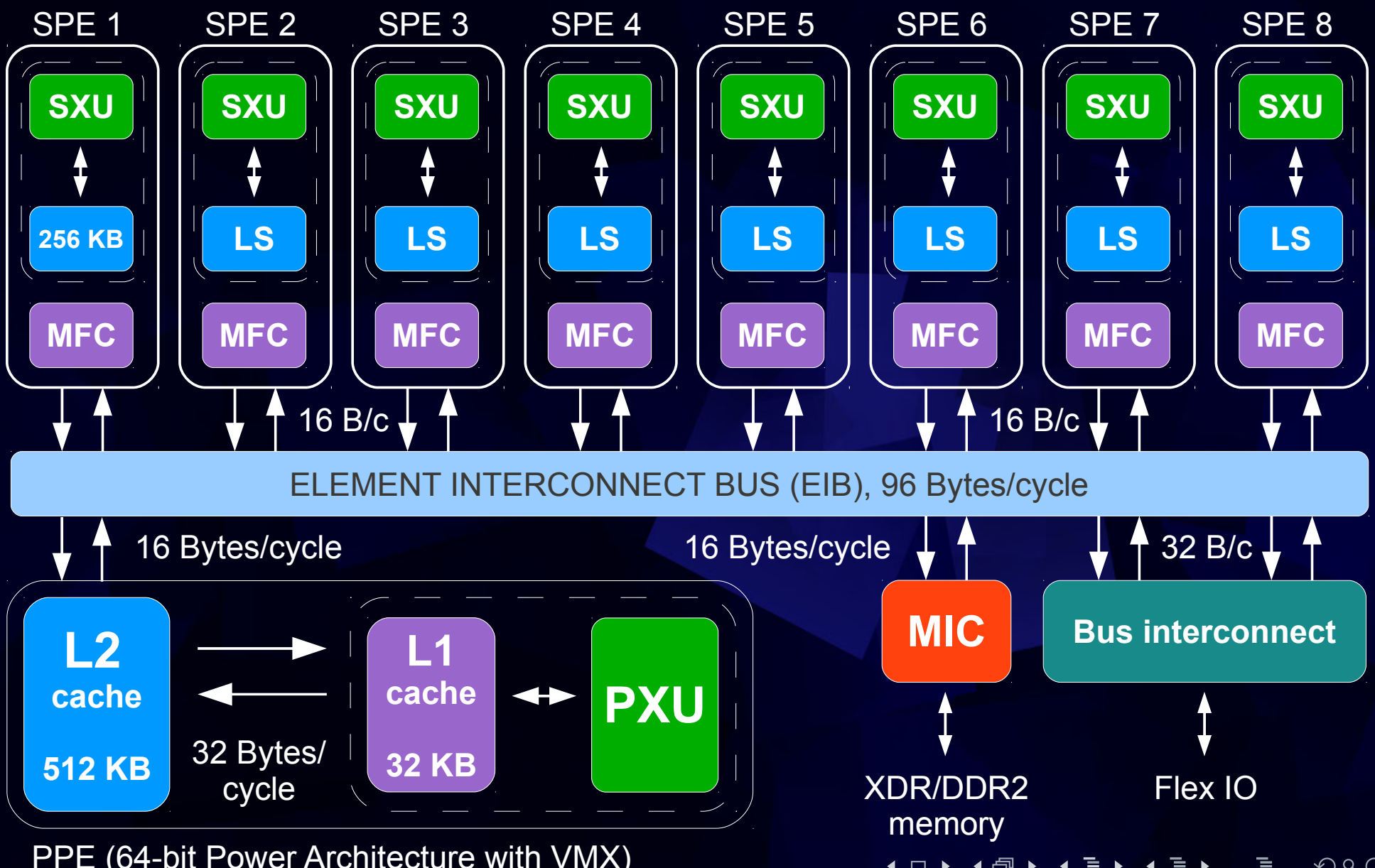
Гибридная система



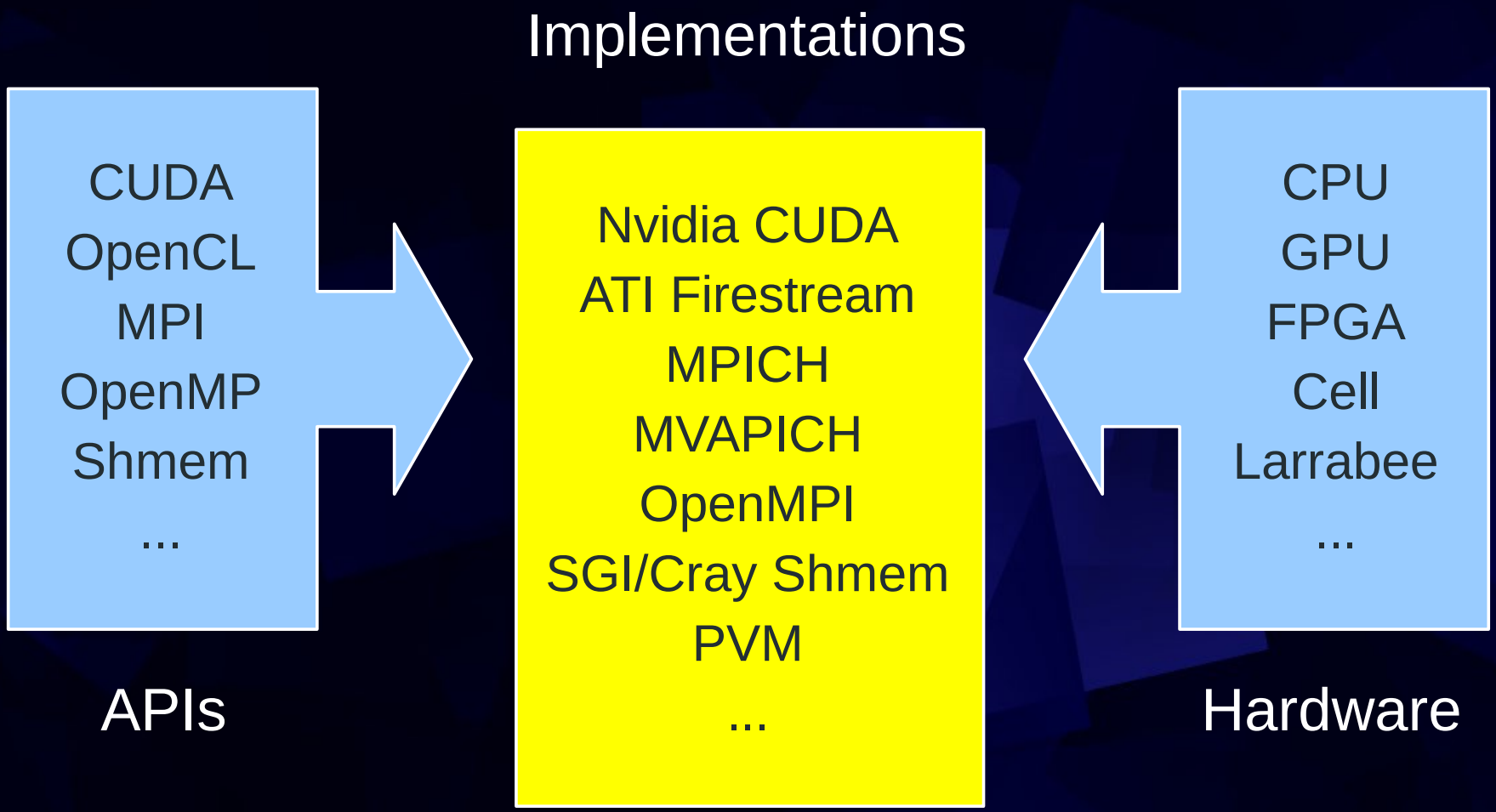
Tesla 10 Architecture



Cell Broadband Engine Architecture



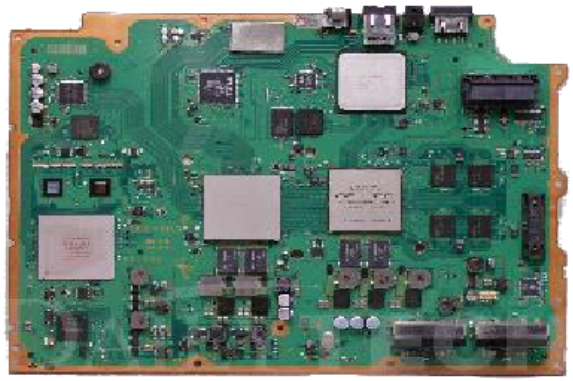
Устройства, стандарты, реализации



Основные свойства операций обмена данными

- Индивидуальные/коллективные
- Одно/двухсторонние (put/get, send/recv)
- Блокирующие/неблокирующие (синхронные/асинхронные)
- Буферизуемые (латентность)
- Упорядоченные
- Фильтруемые (теги сообщений)

Cell Broadband Engine



Cell Broadband Engine

Ключевые особенности

- PowerPC-совместимое ядро Power Processing Unit (PPU)
- Возможность установки полноценной операционной системы
- Поддержка векторной арифметики (Altivec/VMX)

Software development kit

- Sony Playstation 3 SDK
- IBM Cell Broadband Engine SDK 3.1
  - Компиляторы GNU и IBM
  - Отладчик GNU GDB
  - Трассировщик (IBM)
  - Прикладные библиотеки (FFT, BLAS, ...)

\+ Компилятор IBM OpenCL для Cell

Cell — программа для PPU

```
 1 #include <pthread.h>
 2
 3 #define ALIGN 16
 4
 5 ...
 6
 7 int main(int argc, char* argv[])
 8 {
 9         const int nspe = 3;                     // Number of SPEs to use.
10
11         char args[][ALIGN] __attribute__((aligned(ALIGN))) =
12                 { "red", "green", "blue" };     // Define aligned arguments array.
13
14         pthread_t threads[nspe];                // Array to keep threads handles.
15
16         int ispe;                               // Start SPE threads.
17         for (ispe = 0; ispe < nspe; ispe++)
18                 pthread_create(&threads[ispe], NULL, &spe_thread, args[ispe]);
19
20         for (ispe = 0; ispe < nspe; ispe++)     // Wait for SPE threads to finish.
21                 pthread_join(threads[ispe], NULL);
22
23         return 1;
24 }
```

Cell — POSIX-поток для SPE-контекста

```
 1 // Cell SPE library header.
 2 #include <libspe2.h>
 3
 4 #define ULL unsigned long long
 5
 6 // SPU program handle used by context loader.
 7 extern spe_program_handle_t sample_spu;
 8
 9 void* spe_thread(void* arg)
10 {
11         int flags = 0;
12         unsigned int entry = SPE_DEFAULT_ENTRY;
13
14         char* name = (char*)arg;        // Extract thread parameter.
15
16         ULL argp = (ULL)(char*)arg;     // Parameters to pass to SPE main entry.
17         ULL envp = strlen(name);
18
19         // Create SPE context and load SPE program
20         // into this context.
21         spe_context_ptr_t context = spe_context_create(0, NULL);
22         spe_program_load(context, &sample_spu);
23
24         // Run SPE context, i.e. execute SPE program.
25         spe_context_run(context, &entry, flags, (void*)argp, (void*)envp, NULL);
26
27         pthread_exit(NULL);
28
29         return NULL;
30 }
31
```

Cell — Synergistic Processing Unit (SPU)

Ключевые особенности

- Возможность исполнения на SPE обычной последовательной программы
- Локальная память SPE для команд и данных – 256 Кбайт
- Явное управление обменом данных на SPE
- Отдельные конвейеры исполнения для арифметики и пр. команд
- Векторная арифметика на 128-битовых регистрах
- Большое число регистров

Ограничения

- Высокая латентность DMA-операций
- На SPE нет аппаратной поддержки деления
- Задержки в конвейерах исполнения SPE
- TLB-промахи без использования больших страниц памяти
- Ограниченная длина очереди DMA-запросов
- Задержки при отсутствии выравнивания DMA по 128 байт

Cell — программа для SPU

```
 1 // SPE specific I/O routines header.
 2 #include <spu_mfcio.h>
 3
 4 int main(ULL id, ULL argp, ULL envp)
 5 {
 6         // Define aligned local char array.
 7         char name[ALIGN] __attribute__((aligned(ALIGN)));
 8
 9         unsigned int tag = mfc_tag_reserve();          // Reserve MFC tag.
10
11         mfc_get(name, argp, ALIGN, tag, 0, 0);         // Initiate a data transfer
12                                                        // from main memory address
13                                                        // argp to local storage
14                                                        // address name with size
15                                                        // ALIGN and operation tag
16
17         mfc_write_tag_mask(1 << tag);                  // Wait for data transfer
18         mfc_read_tag_status_all();                     // to acomplish.
19
20         mfc_tag_release(tag);                          // Release MFC tag.
21
22         return 1;
23 }
```

MPI — установка и настройка

Install MPI package

```
yum install mpich2 mpich2-devel
yum install openmpi openmpi-devel
```

Create .mpd.conf (/etc/mpd.conf)

```
MPD_SECRETWORD=<password>
```

```
chmod 600 .mpd.conf
```

Create mpd.hosts files:

MPICH:

```
hostname:<num_cores>
...
hostname:<num_cores>
```

OpenMPI:

```
hostname slots=<num_cores>
...
hostname slots=<num_cores>
```

MPI hello_world.c

```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        const int seconds = 10;

        int ierr = MPI_Init(&argc, &argv);
        if (ierr != MPI_SUCCESS)
                printf("Failed to initialize MPI\n");

        int size;
        ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (ierr != MPI_SUCCESS)
                printf("Failed to retrieve MPI communicator size\n");

        int rank;
        ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if (ierr != MPI_SUCCESS)
                printf("Failed to retrieve MPI communicator rank\n");

        printf("MPI node %d initialized\n", rank);

        sleep(seconds);

        printf("MPI node %d finished\n", rank);

        ierr = MPI_Finalize();

        return 0;
}
```

MPI hello_world.f90

```
 1 program main
 2   use ISO_C_BINDING
 3   implicit none
 4   include 'mpif.h'
 5
 6   interface
 7     function sleep(seconds) bind(C)
 8       use ISO_C_BINDING
 9       implicit none
10       integer(C_INT), value :: seconds
11       integer :: sleep
12     end function sleep
13   end interface
14
15   integer :: ierr, szcomm, rank
16   integer, parameter :: seconds = 10
17
18   call MPI_Init(ierr)
19   if (ierr .ne. MPI_SUCCESS) then
20     print *, 'Failed to initialize MPI'
21   endif
```

```
22
23   call MPI_Comm_size(MPI_COMM_WORLD, szcomm)
24   if (ierr .ne. MPI_SUCCESS) then
25     print *, 'Failed to retrieve MPI communicator size'
26   endif
27
28   call MPI_Comm_rank(MPI_COMM_WORLD, rank)
29   if (ierr .ne. MPI_SUCCESS) then
30     print *, 'Failed to retrieve MPI communicator rank'
31   endif
32
33   print '("MPI node ", i1, " initialized")', rank
34
35   ierr = sleep(seconds)
36
37   print '("MPI node ", i1, " finished")', rank
38
39   call MPI_Finalize(ierr)
40
41 end program main
```

MPICH2 — управление

Launch mpd daemon on each node

```
$ mpdboot -v
LAUNCHED mpd on msiwind  via
mpdboot_msiwind (handle_mpd_output 420): from mpd on
msiwind, invalid port info: no_port

$ mpd &
$ mpdallexit
$ mpdboot
LAUNCHED mpd on msiwind via
RUNNING: mpd on msiwind
```

- Multiple mpd instances per node
- Multicore CPUs:

```
$ mpdboot --totalnum=-1 --ncpus=2
```

Execute MPI program:

```
$ mpirun -np <num_processes> <executable>
```

Shutdown mpd daemons

```
$ mpdallexit
```

Обращение матрицы методом Монте-Карло

Марковские цепочки для вектора $x = (x_1 \dots x_m)$:

$$k_0 \to k_1 \to \dots \to k_i$$

где m — число всевозможных состояний

Начальные вероятности и вероятности переходов:

$$P\{k_0 = \alpha\} = p_\alpha,$$

$$P\{k_j = \beta | k_{j-1} = \alpha\} = p_{\alpha\beta}$$

Обращение матрицы методом Монте-Карло

Марковские цепочки для матрицы $A = (a_{ij}), n \times n$:

$$r \to k_0 \to k_1 \to \ldots \to k_i$$

Весовые коэффициенты марковской цепи для матрицы A:

$$W_j = \frac{a_{k_0 k_1} a_{k_1 k_2} \cdots a_{k_{j-1} k_j}}{p_{k_0 k_1} p_{k_1 k_2} \cdots p_{k_{j-1} k_j}}$$

или рекуррентно:

$$W_j = W_{j-1} \frac{a_{k_{j-1} k_j}}{p_{k_{j-1} k_j}}, W_0 = 1$$

где $p_{k_{j-1} k_j}$ — вероятности перехода

Обращение матрицы методом Монте-Карло

Задача: дана матрица B, $n \times n$, найти $\mathrm{C} : BC = I$.

Метод Монте-Карло:

$$c_{rr'} \approx \frac{1}{N} \sum_{s=1}^{N} \left[\sum_{(j|k_j=r')} W_j \right]$$

где $W_j = W_{j-1} \frac{a_{k_{j-1}k_j}}{p_{k_{j-1}k_j}}$, $W_0 = 1$, $A = I - B$,

$(j|k_j = r')$ — учитываются лишь те W_j, для которых $k_j = r'$,

$r, r' = 1, 2, \ldots, n$, N - число используемых марковских цепочек.

"Почти"оптимальные вероятности перехода
(Megson, Aleksandrov, Dimov, 1994):

$$p_{\alpha\beta} = \frac{|a_{\alpha\beta}|}{\sum_\beta |a_{\alpha\beta}|}$$

Обращение матрицы методом Монте-Карло

```
do i = imin, imax
  do j = jmin, jmax
    C(i,j) = 0.0
    if (i .eq. j) C(i,j) = nchains
    do s = 1, nchains
      chain0 = i
      w0 = 1.0
      do
        call random_number(val)
        jj = 1
        do
          val = val - abs(A(chain0, jj)) / Sa(chain0)
          if (val .lt. 0.0 .or. jj .eq. n) exit
          jj = jj + 1
        enddo
        chain1 = jj

        w1 = w0 * sign(Sa(chain0), A(chain0, chain1))

        if (chain1 .eq. j) then
          C(i,j) = C(i,j) + w1
        endif
        chain0 = chain1
        w0 = w1

        if (abs(w1) .lt. delta) exit
      enddo
    enddo
    C(i,j) = C(i,j) / nchains
  enddo
enddo
```

Обращение матрицы методом Монте-Карло

Оценка сложности:

$$O\left(n^2 NT\right)$$

где N — число марковских цепочек,
T — мера длины цепочек,
не зависящие от размерности матрицы

Оценка на длину цепочек (Megson, Aleksandrov, Dimov, 1994):

$$N = \left(\frac{0.6745}{\varepsilon(1 - ||A||)}\right)^2$$

где ε — заданная погрешность результата

Обращение матрицы — MPI

```
 1   include 'mpif.h'
 2
 3   ...
 4
 5   call MPI_Bcast(B, n * n, MPI_REAL, root, MPI_COMM_WORLD, ierr)
 6   call MPI_Barrier(MPI_COMM_WORLD, ierr)
 7
 8   call MatInverseMC(B, C, n, eps, delta, imin, imax, jmin, jmax)
 9
10   if (rank .ne. root) then
11     nnpart = (imax - imin + 1) * (jmax - jmin + 1)
12     call MPI_Send(C(imin, jmin), nnpart, MPI_REAL, root, rank,
       MPI_COMM_WORLD, ierr)
13   else
14     do i = 0, root - 1
15       imin = 1
16       imax = n
17       jmin = i * npart + 1
18       jmax = jmin + npart - 1
19       nnpart = (imax - imin + 1) * (jmax - jmin + 1)
20       call MPI_Recv(C(imin, jmin), nnpart, MPI_REAL, i, i,
         MPI_COMM_WORLD, rstatus, ierr)
21     enddo
22   endif
```

Обращение матрицы — OpenMP

```
 1 int npart = n / nthreads;
 2
 3 #pragma omp parallel for
 4 for (int ithread = 0; ithread < nthreads; ithread++)
 5 {
 6         int imin = 1;
 7         int imax = n;
 8         int jmin = ithread * npart + 1;
 9         int jmax = imin + npart - 1;
10
11         if (ithread == nthreads - 1)
12                 jmax = n;
13
14         matinversemc(B, C, &n, &eps,
15                 &delta, &imin, &imax, &jmin, &jmax);
16 }
17
```

Обращение матрицы — Boost::thread

```
 1 #include <boost/thread/thread.hpp>
 2 #include <boost/thread/mutex.hpp>
 3
 4 ...
 5
 6 // Create threads
 7 int npart = n / nthreads;
 8 int imin = 1, imax = n;
 9 int jmin[nthreads], jmax[nthreads];
10 boost::thread_group threads;
11 for (int ithread = 0; ithread < nthreads - 1; ithread++)
12 {
13         jmin[ithread] = ithread * npart + 1;
14         jmax[ithread] = jmin[ithread] + npart - 1;
15
16         threads.create_thread(boost::bind(&matinversemc,
17                 B, C, &n, &eps, &delta, &imin, &imax,
18                 jmin + ithread, jmax + ithread));
19 }
20
21 // Last thread
22 {
23         jmin[nthreads - 1] = (nthreads - 1) * npart + 1;
24         jmax[nthreads - 1] = n;
25
26         matinversemc(B, C, &n, &eps, &delta, &imin, &imax,
27                 jmin + nthreads - 1, jmax + nthreads - 1);
28 }
29
30 // Wait for threads to finish
31 threads.join_all();
```

Заключение

- CUDA встраивается в существующие программные модели гибридных вычислений
- Взаимодействие со множеством API, на различных языках $\Rightarrow$ возможно портирование на CUDA частей существующих параллельных приложений