Rock’ em Graphic Cards

Agnes Meyder

27.12.2013, 16:00
Layout

Motivation

Parallelism

Old Standards
   OpenMPI
   OpenMP

Accelerator Cards
   CUDA
   OpenCL
   OpenACC
   Hardware
   C++AMP

The End
Layout

Motivation

Parallelism

Old Standards
  OpenMPI
  OpenMP

Accelerator Cards
  CUDA
  OpenCL
  OpenACC
  Hardware
  C++AMP

The End
Buzz word for: “Pattern Recognition in large Datasets“
Figure: hERG - Possible Opening Movement¹

¹http://dx.doi.org/10.1016/j.bmcl.2005.01.008
Sponsors for my Summer School

Bundesministerium für Bildung und Forschung

Universität Hamburg

Bayer CropScience

Science For A Better Life
Layout

Motivation

Parallelism

Old Standards
OpenMPI
OpenMP

Accelerator Cards
CUDA
OpenCL
OpenACC
Hardware
C++AMP

The End
Parallelism

<table>
<thead>
<tr>
<th>Data-based</th>
<th>Task-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>OpenMP</td>
</tr>
<tr>
<td>OpenCL</td>
<td>OpenMPI</td>
</tr>
<tr>
<td>C++AMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>data-based</td>
<td>task-based</td>
</tr>
</tbody>
</table>
Types of Parallelism

**Data-Parallelism**
- Fry omelets for 8,000 people.

**Task-Parallelism**
- Cook a 5-course menu for one person.

Cook a menu for 8,000 people with an omelet as part of the dessert.
PP - Possible Problems

- Kitchen too small
- Only apprentices available
- Not enough frying pans
- Delivery paths too small (only one can access the fridge)
- Only one can write a new recipe into the book
- Transport 30 eggs in one go
- Serve the courses in the correct order
PP - Possible Problems

- Kitchen too small ⇒ Global capacity limitation
- Only apprentices available ⇒ Processor complexity limited
- Not enough frying pans ⇒ Concurrent task number limited
- Delivery paths too small (only one can access the fridge) ⇒ Band width limitation
- Only one can write a new recipe into the book ⇒ Read-write access limitation
- Transport 30 eggs in one go ⇒ coalescing memory access
- Serve the courses in the correct order ⇒ Synchronization issues
Layout

Motivation

Parallelism

Old Standards
  OpenMPI
  OpenMP

Accelerator Cards
  CUDA
  OpenCL
  OpenACC
  Hardware
  C++AMP

The End
Parallelism - Old Standards

<table>
<thead>
<tr>
<th></th>
<th>data-based</th>
<th>task-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>explicit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OpenMP

OpenMPI
OpenMPI is one implementation of the standard MPI.

▶ Install openmpi and its devel packages
▶ Switch linker and compiler to mpicc
▶ Compile: mpicc -I/usr/lib64/mpi/gcc/openmpi/include ...
▶ Link: mpicc -o "openmpi" ./src/openmpi.o -lmpi
▶ Run code with: mpirun -np 1 -hostfile hostfile ./openmpi

hostfile: a file with the names of usable hosts. “localhost” is fitting for testing purposes.

Documentation: http://www.open-mpi.org/
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process %d on %s out of %d\n",
          rank, processor_name, numprocs);

  // cook one course per task ?

  MPI_Finalize();
}
```

2 http://www.linux-mag.com/id/5759/
Typical (Parallel) Task: Matrix-Multiplication
Matrix-Multiplication

\[ C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj} \]

https://en.wikipedia.org/wiki/Matrix_multiplication
Matrix-Multiplication

```c
const int NUM_ROWS_A = 200;
const int NUM_COLS_A = 300;
const int NUM_ROWS_B = NUM_COLS_A;
const int NUM_COLS_B = 400;

int *A = fillMatrix(NUM_ROWS_A, NUM_COLS_A);
int *B = fillMatrix(NUM_ROWS_B, NUM_COLS_B);
int *C = (int*) malloc(sizeof(int)*NUM_ROWS_A * NUM_COLS_B);

for (unsigned int i = 0; i < NUM_ROWS_A; i++)
{
    for (unsigned int j = 0; j != NUM_COLS_B; ++j)
    {
        C[i* NUM_COLS_B + j] = 0;
        for (unsigned int k = 0; k != NUM_COLS_A; ++k)
        {
            C[i* NUM_COLS_B + j] += A[i* NUM_COLS_A + k] * B[k* NUM_COLS_B + j];
        }
    }
    // Do what you need with C ...
}
free(A); free(B); free(C);
```
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void) {
    // everything in the bracket is parallel!
    #pragma omp parallel
    {
        printf("Hello World !! This program has %d threads.. ",
               "This line is printed by Thread: %d \n",
               omp_get_num_threads(), omp_get_thread_num());
    }
    return EXIT_SUCCESS;
}
```

---

4 http://www.linux-mag.com/id/5759/
# include <stdio.h>
# include <stdlib.h>
# include <omp.h>

int * fillMatrix(const int num_rows, const int num_cols)
{
    int i;
    int *pointer = (int*)malloc(sizeof(int)*num_cols * num_rows);
    for (i = 0; i < num_cols * num_rows; i++) {
        pointer[i] = i;
    }
    return pointer;
}

int main(void) {
    const int NUM_ROWS_A = 2;
    const int NUM_COLS_A = 3;
    const int NUM_ROWS_B = NUM_COLS_A;
    const int NUM_COLS_B = 4;

    int *A = fillMatrix(NUM_ROWS_A, NUM_COLS_A);
    int *B = fillMatrix(NUM_ROWS_B, NUM_COLS_B);
    int *C = (int*)malloc(sizeof(int)*NUM_ROWS_A * NUM_COLS_B);
}
OpenMP - Matrix-Multiplication II

```c
// everything in the bracket is parallel!
#pragma omp parallel
{
    printf("Hello World! This program has %d threads.. ", omp_get_num_threads());
    printf("This line is printed by Thread: %d \n", omp_get_thread_num());
    for(unsigned int i = 0; i != NUM_ROWS_A; ++i)
    {
        for(unsigned int j =0; j != NUM_COLS_B; ++j)
        {
            C[i* NUM_COLS_B +j] = 0;
            for(unsigned int k=0; k != NUM_COLS_A; ++k)
            {
                C[i* NUM_COLS_B +j] += A[i* NUM_COLS_A +k ] * B[k* NUM_COLS_A +j ];
                printf( "C[%d, %d] is calculated by Thread: %d \n", i, j, omp_get_thread_num());
            }
        }
    }
} // end pragma
```
for (unsigned int i = 0; i != NUM_ROWS_A; ++i)
{
    for (unsigned int j = 0; j != NUM_COLS_B; ++j)
    {
        printf( "Row: %d Column: %d Value: %d\n", i, j, C[i*NUM_COLS_B + j]);
    }
}
free(A);
free(B);
free(C);
return EXIT_SUCCESS;
Hello World! This program has 4 threads..
This line is printed by Thread: 2
C[0, 0] is calculated by Thread: 2
...
Hello World! This program has 4 threads..
This line is printed by Thread: 3
C[0, 0] is calculated by Thread: 3
...
Hello World! This program has 4 threads..
This line is printed by Thread: 0
Hello World! This program has 4 threads..
This line is printed by Thread: 1
C[0, 0] is calculated by Thread: 1
...
int i, j, k, chunk;
...
const int CHUNKSIZE = 4;
...
chunk = CHUNKSIZE;

#pragma omp parallel shared(A,B,C,chunk,i) private(j, k)
{
    printf("Numer of launched threads: %d\n", 
            omp_get_num_threads());
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < NUM_ROWS_A; i++)
    {
        for(j=0; j != NUM_COLS_B; ++j)
        {
            C[i*NUM_COLS_B +j] = 0;
            for(k=0; k != NUM_COLS_A; ++k)
            {
                C[i*NUM_COLS_B +j] += A[i*NUM_COLS_A +k ]
                * B[k*NUM_COLS_B +j ];
            }
        } /* end of parallel for loop */
    } /* end of parallel section */
OpenMP - Installation Sprint

- Install libgomp
- Compile: gcc -fopenmp ...
- Link: gcc -o "openmp" ./src/openmp.o -lgomp

Documentation: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
Layout

Motivation

Parallelism

Old Standards
  OpenMPI
  OpenMP

Accelerator Cards
  CUDA
  OpenCL
  OpenACC
  Hardware
  C++AMP

The End
Parallelism

- **data-based**
  - CUDA
  - OpenCL

- **task-based**
  - OpenMPI

- **explicit**
- **implicit**

OpenMP
Accelerator Terminology

Some words:
- Host = master thread, often CPU
- Device = helper threads, often on accelerator cards
- Kernel = functions, running on the device

Some words:
- In order to run OpenCL- or CUDA-Code, please install the proprietary drivers for your graphic card!
CUDA- Matrix Multiplication I

```c
#include <stdio.h>
#include <stdlib.h>

__global__ void matrixMultiply(int *A, int *B, int *C,
    int NUM_ROWS_A, int NUM_COLS_A,
    int NUM_ROWS_B, int NUM_COLS_B,
    int NUM_ROWS_C, int NUM_COLS_C) {
    int row = blockIdx.x * blockDim.x + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((row < NUM_ROWS_C) && (col < NUM_COLS_C)) {
        int sum = 0;
        for (int k = 0; k < NUM_COLS_A; ++k) {
            sum += A[row*NUM_COLS_A + k] * B[k*NUM_COLS_B + col];
        }
        C[row*NUM_COLS_C+col] = sum;
    }
}

int main(void) {
```
int *A = fillMatrix(NUM_ROWS_A, NUM_COLS_A);
int *B = fillMatrix(NUM_ROWS_B, NUM_COLS_B);
int *C = (int*)malloc(sizeof(int)*NUM_ROWS_C * NUM_COLS_C);
for(int i = 0; i != NUM_ROWS_C * NUM_COLS_C; ++i)
{
    C[i] = -1;
}
cudaMalloc((void**) &dA, NUM_ROWS_A*NUM_COLS_A*sizeof(int));
cudaMalloc((void**) &dB, NUM_ROWS_B*NUM_COLS_B*sizeof(int));
cudaMalloc((void**) &dC, NUM_ROWS_C*NUM_COLS_C*sizeof(int));
cuMemcpy(dA, A, NUM_ROWS_A*NUM_COLS_A*sizeof(int), cudaMemcpyHostToDevice);
cuMemcpy(dB, B, NUM_ROWS_B*NUM_COLS_B*sizeof(int), cudaMemcpyHostToDevice);

// Initialize the grid and block dimensions here
dim3 dimGrid(1, 1,1);
dim3 dimBlock(NUM_COLS_C, NUM_ROWS_C, 1);

// Launch the GPU Kernel here
matrixMultiply<<<dimGrid, dimBlock>>>(dA, dB, dC, NUM_ROWS_A, NUM_COLS_A, NUM_ROWS_B, NUM_COLS_B, NUM_ROWS_C, NUM_COLS_C);
cudaThreadSynchronize();

// Copy the GPU memory back to the CPU here
cudaMemcpy(C, dC, NUM_ROWS_C*NUM_COLS_C*sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(dC);
cudaFree(dB);
cudaFree(dA);

// Do what you need with C ...

free(A);
free(B);
free(C);
return EXIT_SUCCESS;
__global__ void matrixMultiply(int *A, int *B, int *C, int NUM_ROWS_A, int NUM_COLS_A,
int NUM_ROWS_B, int NUM_COLS_B, int NUM_ROWS_C, int NUM_COLS_C) {
    int row = blockIdx.x * blockDim.x + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((row < NUM_ROWS_C) && (col < NUM_COLS_C)) {
        int sum = 0;
        for (int k = 0; k < NUM_COLS_A; ++k) {
            sum += A[row * NUM_COLS_A + k] * B[k * NUM_COLS_B + col];
        }
        C[row * NUM_COLS_C + col] = sum;
    }
}

int main(void) {
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(NUM_COLS_C, NUM_ROWS_C, 1);

    matrixMultiply<<<dimGrid, dimBlock>>>(dA, dB, dC, NUM_ROWS_A, NUM_COLS_A,
NUM_ROWS_B, NUM_COLS_B, NUM_ROWS_C, NUM_COLS_C);
    cudaThreadSynchronize();
}
CUDA - Installation Sprint

- Install CUDA SDK and your os CUDA package (check compatibility of gcc!)
- Compile: `nvcc ... --compiler-bindir /path/to/special/gcc`
- Link: `gcc -L/usr/local/cuda/lib64 ... -lcudart`

**Documentation:**  [http://docs.nvidia.com/cuda/](http://docs.nvidia.com/cuda/)
```c
#include <stdio.h>
#include <stdlib.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main(void) {
    // Create the two input vectors
    int i;
    const int LIST_SIZE = 1024;
    int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
    int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
    for (i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }
}

```
// Load the kernel source code into the array source_str
FILE *fp;
char *source_str;
size_t source_size;
fp = fopen("src/vector_add_kernel.cl", "r");
if (!fp) {
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

// Get platform and device information
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,
                      &device_id, &ret_num_devices);
// Create an OpenCL context
cl_context context = clCreateContext( NULL, 1, &device_id,
        NULL, NULL, &ret);

// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue(
        context, device_id, 0, &ret);

// Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
        LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
        LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        LIST_SIZE * sizeof(int), NULL, &ret);

// Copy the lists A and B to their respective memory buffers
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj,
        CL_TRUE, 0, LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj,
        CL_TRUE, 0, LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,
        (const char **)&source_str, (const size_t *)&source_size,
        &ret);
// Build the program
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

// Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

// Set the arguments of the kernel
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

// Execute the OpenCL kernel on the list
size_t global_item_size = LIST_SIZE; // Process the entire lists
size_t local_item_size = 64; // Process in groups of 64
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, NULL, &global_item_size, &local_item_size, 0, NULL, NULL);
Read the memory buffer C on the device to the local variable C

```c
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue, c_mem_obj, 
    CL_TRUE, 0, LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
```

Display the result to the screen

```c
for (i = 0; i < LIST_SIZE; i++)
    printf("%d + %d = %d\n", A[i], B[i], C[i]);
```

Clean up

```c
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(a_mem_obj);
ret = clReleaseMemObject(b_mem_obj);
ret = clReleaseMemObject(c_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
free(A);
free(B);
free(C);
return 0;
```
// Get platform and device information
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,
    &device_id, &ret_num_devices);
...

// Execute the OpenCL kernel on the list
size_t global_item_size = LIST_SIZE; // Process the entire lists
size_t local_item_size = 64; // Process in groups of 64
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
    NULL, &global_item_size, &local_item_size, 0, NULL, NULL);
...

// Read the memory buffer C on the device to the local variable C
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue, c_mem_obj,
    CL_TRUE, 0, LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
OpenCL - Installation Sprint

- Install Khronos OpenCL-Headers and if necessary SDK of AMD/NVIDIA
- Compile: gcc ...
- Link: gcc ... -lOpenCL

Documentation:
http://www.khronos.org/opencl/
General Code Structure

- Define accelerator device
- Initialize: memory on host and device
- Specify amount and dimensions of necessary kernel launches
- Launch Kernel
- Cleanup: Store result, free data
```c
#pragma omp parallel shared(A,B,C,chunk, i) private(j, k) 
{
    printf("Number of launched threads: %d\n", 
    omp_get_num_threads());

#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < NUM_ROWS_A; i++)
{
    for(j=0; j != NUM_COLS_B; ++j)
    {
        C[i*NUM_COLS_B +j] = 0;
        for(k=0; k != NUM_COLS_A; ++k)
        {
            C[i*NUM_COLS_B +j] += A[i*NUM_COLS_A +k ]
            * B[k*NUM_COLS_B +j ];
        }
    }
} /* end of parallel for loop */
} /* end of parallel section */
```
OpenACC - Matrix-Multiplication

**OpenMP:**

```c
#pragma omp parallel shared(A,B,C,chunk, i) private(j, k)
{
   #pragma omp for schedule(dynamic,chunk) nowait
   for (i=0; i < NUM_ROWS_A; i++)
   {

   }
```

**OpenACC:**

```c
#pragma acc parallel
{
   #pragma acc loop
   for (i=0; i < NUM_ROWS_A; i++)
   {

   }
```

```c
for(j=0; j != NUM_COLS_B; ++j)
{
   C[i*NUM_COLS_B +j] = 0;
for(k=0; k != NUM_COLS_A; ++k) { ...
```
OpenACC - Installation Sprint?

Already available in cray, cups, pgi compilers.

Available in gcc in 2015?!
Tiled Matrix Multiplication I

\[ C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj} \]
Tiled Matrix Multiplication II

$$C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$$
Tiled Matrix Multiplication - CUDA Warp Divergence

\[ C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj} \]
NVIDIA\textsuperscript{7} - Simplified Architecture Scheme

\textsuperscript{7}http://www.nvidia.com
## Thread vs. Block vs. Grid Dimensions in CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>work-item</td>
<td>vector (worker)</td>
</tr>
<tr>
<td>block</td>
<td>work-group</td>
<td>worker / gang (gang)</td>
</tr>
<tr>
<td>grid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![3D diagram showing thread, block, and grid dimensions](image-url)
Hardware

- Graphic Cards (AMD, NVIDIA)
- Xeon Phi (Intel)
- parallela (http://parallela.org)
- FPGAs (field programmable gate arrays)
NVIDIA Tesla card\(^8\) - 2000€

\(^8\)http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
Intel Xeon Phi⁹ - 2000€

Parallela$^{10}$ - $99$

$^{10}$http://parallela.org
Parallelism

- **data-based**
  - CUDA
  - OpenCL
  - C++AMP

- **task-based**
  - OpenMPI
  - OpenMP
  - OpenACC

Emphasis on languages with C-Bindings today.
Layout

Motivation

Parallelism

Old Standards
  OpenMPI
  OpenMP

Accelerator Cards
  CUDA
  OpenCL
  OpenACC
  Hardware
  C++AMP

The End
I did not mention:

- POSIXThreads
- BOOST Threads
- Cilk (in C++: Cilk Plus)
- Intel Thread Building Blocks
- task-scheduling programs such as StarPU, oomps

... and languages such as Python with rudimentary multiprocessing abilities.
Parallelism

<table>
<thead>
<tr>
<th>data-based</th>
<th>task-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>OpenMPI</td>
</tr>
<tr>
<td>C++AMP</td>
<td>OpenMP</td>
</tr>
<tr>
<td>OpenACC</td>
<td></td>
</tr>
</tbody>
</table>