Supercomputing for Everyone Series: Performance Tuning Summer School
L09: An Introduction to GPGPU Programming

Dr. Guido Juckeland, Dr. Holger Brunst
Technische Universität Dresden, Germany

August 17-21, 2015
Organization of this class

- course runs August 17-21, from 11 a.m. until 4.30 p.m. (EDT)
- 20 sites, each site should have an instructor/TA/helper
- please put your microphones on mute, if you are not asking questions
- set up in lectures (60 min) and exercises (75 min)
- Q&A time at end of each lecture
- chat for discussions
- class material and links: http://go.iu.edu/CtB
Lunch breaks are important

Monday
11.00-12.30  L1_Performance is ambiguous
12.30-13.15  X1_First steps on Blue Waters
13.15-14.15  Lunch break
14.15-15.15  L2_Evaluation first!
15.15-16.30  X2_Pen, paper, and performance

Tuesday
11.00-12.00  L3_Use simple tools for simple questions
12.00-13.15  X3_The command line is your friend
13.15-14.15  Lunch break
14.15-15.15  L4_Tuning needs persistence
15.15-16.30  X4_Benchmarks provide the baseline

Wednesday
11.00-12.00  L5_Core tuning pays off n-times
12.00-13.15  X5_How to access data efficiently
13.15-14.15  Lunch break

Thursday
14.15-15.15  L6_Visual tools can be fun, sometimes
15.15-16.30  X6_Zoom and scroll

Friday
11.00-12.00  L7_Sharing may double the sorrow (OpenMP)
12.00-13.15  X7_OpenMP enables quick 'n easy gains
13.15-14.15  Lunch break
14.15-15.15  L8_Hand made parallelization hurts (MPI)
15.15-16.30  X8_Formal MPI to good MPI

* All times are EDT
GPU system setup

CPU

Main Memory

System Bus

Accelerator

Local Memory

DMA Transfers

Application

calls

Accelerator Library

invokes

Subprogram
Defining terms: HOST and DEVICE

HOST

DEVICE
Allocating and freeing memory on the device

```c
int main( int argc, char *argv[] ) {

    float *devicepointer;     //Pointer to float element

    // The following line allocates memory for one float on the GPU and
    // sets devicepointer to the beginning of that memory area
    cudaMalloc( (void**)&devicepointer, sizeof(float) );
    // The following line releases the allocated memory for devicepointer
    // on the GPU so that it may be used again by another allocation
    cudaFree( devicepointer );

    return 0;
}
```
#define NELEMENTS 16
int main( int argc, char *argv[] ) {

    float hostvariable[NELEMENTS]; //float-array on the HOST
    float *devicepointer;         //allocated pointer to the DEVICE

    cudaMemcpy( devicepointer,   //Pointer to DEVICE memory
                 hostvariable,   //Pointer to host memory
                 sizeof(float)*NELEMENTS,   //number of bytes to transfer
                 cudaMemcpyHostToDevice ); //direction of transfer

    return 0;
}

Copying data from the DEVICE to the HOST

#define NELEMENTS 16
int main( int argc, char *argv[] ) {

    float hostvariable[NELEMENTS];    //float-array on the HOST
    float *devicepointer;           //allocated pointer to the DEVICE

    cudaMemcpy( hostvariable,     //Pointer to HOST memory
                devicepointer,     //Pointer to DEVICE memory
                sizeof(float) )*NELEMENTS,  //number of bytes to transfer
                      cudaMemcpyDeviceToHost ); //direction of transfer

    return 0;
}

Compiling and running a CUDA program

- On remote systems make sure that the CUDA environment is available (usually requires a `module load cudatoolkit` or similar)
- Name your CUDA files with the suffix `.cu`
- Compile your program using `nvcc` (e.g. `nvcc myprogram.cu`)
- Execute your program by running `./a.out`
- On remote systems the node you compile on might not feature a GPU, you will have to use a batch system to access the “right” node.
The five basic CUDA functions

- cudaMalloc to allocate memory on the device
- cudaMemcpy to transfer data to and from the device
- Kernel invocations (we will cover this later)
- Handling errors
- cudaFree to release allocated memory on the device
Kernels

A **kernel** is a piece of a program that will be compiled for being executed on the GPU. Kernels are invoked by the host on the device:

```c
int main( int argc, char *argv[] ) {

    kernel1<<<...,...>>>(...);
    kernel2<<<...,...>>>(...);
    kernel3<<<...,...>>>(...);
    return 0;
}
```

- Kernel launches are asynchronous on the host
- Kernel order is sequential on the device
Synchronous behavior

- Synchronous operations wait until the activity is finished
Asynchronous behavior

- Asynchronous Operations launch their activity and return immediately to the calling context.
Kernel declaration

- Kernels are declared like "normal" functions of return type `void` prepended by the key word `__global__`
  Example:
  ```c
  __global__ void do_nothing(float *data) { ... }
  ```

- Since kernels are launched asynchronously they cannot return a value

- Kernels can invoke device functions
  ```c
  __device__ float help_do_nothing() { ... }
  ```
Creating thread blocks

- Order of thread execution is not fixed and can vary
- Threads are executed in batches of 32 threads (warp) in SIMD fashion
- Thread blocks can have arbitrary sizes (within limits, but up to 3D)
- Arrangement of threads is called block

```c
int nx; int ny; int nz; ...
dim3 block(nx, ny, nz); // nx, ny, nz = describes the block in 3D
kernel<<<1,block>>>(...); // creates nx*ny*nz threads in 1 block
Alternative:
kernell<<<1,512>>>(...); // blocksize can also be a number, then 1D
```
The dim3 data structure

```c
struct dim3 {
    unsigned int x, y, z;
};
```

- Create with just assigning a variable, unused dimensions are set to 1
Block size restrictions

- Total number of threads in a block is the product of the number of threads in each dimension
- Total number of threads and threads per dimension have limits

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. block size in x,y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Max. block size in z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Max. threads per block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1024</td>
</tr>
</tbody>
</table>
Multiple thread blocks (a.k.a. grid of blocks)

- Order of block execution is not fixed and can vary
- Multiple blocks can be put onto one multiprocessor (as long as resources are available)
- Blocks are spread over all multiprocessors
- Arrangement of blocks is called grid

```c
int nx; int ny; int nz; ...
dim3 grid(nx, ny, nz); // nx, ny, nz = describes the grid in 3D
kernel<<<grid, block>>>(...); // creates nx*ny*nz blocks
Alternative:
kernel<<<1024, 512>>>(...); // gridsize can also be a number, then 1D
```
Grid size restrictions

- Total number of blocks in a grid is the product of the number of blocks in each dimension
- Total number of blocks and blocks per dimension have limits

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid dimension</td>
<td>2D (grid(nx,ny,1))</td>
<td></td>
<td></td>
<td></td>
<td>3D (grid(nx,ny,nz))</td>
<td></td>
</tr>
<tr>
<td>Max. grid size in x,y,z</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^{31}-1$</td>
</tr>
<tr>
<td>Max. total grid size</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^{31}-1$</td>
</tr>
</tbody>
</table>
Who am I?

- Threads need to decide on which data they need to work
- Requires ID and size queries

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>threadIdx</td>
<td>-</td>
</tr>
<tr>
<td>Block</td>
<td>blockIdx</td>
<td>blockDim</td>
</tr>
<tr>
<td>Grid</td>
<td>-</td>
<td>gridDim</td>
</tr>
</tbody>
</table>

- All variables available in all three dimensions
- Example:
  ```
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  ```
Rule of thumb

• Every thread creates one output element

• Example: Vector-Addition

```c
__global__ void vecadd(int *a, int *b, int *c, int N) {
    // who am I?
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // if I am inside the vector, work on my data
    if ( idx < N ) c[idx] = a[idx] + b[idx];
}
```
3 ways to accelerate applications

Applications

Libraries
“Drop-in” Acceleration

Compiler Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility

OpenACC directive syntax

- **C/C++**
  
  ```c
  #pragma acc directive [clause [,] clause] ...
  ```
  ...often followed by a structured code block

- **Fortran**

  ```fortran
  !$acc directive [clause [,] clause] ...
  ```
  ...often paired with a matching end directive surrounding a structured code block:

  ```fortran
  !$acc end directive
  ```
**SAXPY in C**

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

```
... // Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

---

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i
  !$acc parallel loop
do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$acc end parallel loop
end subroutine saxpy
```

```
... ! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...```

---

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
Compiler directives by example
while ( err > tol && iter < iter_max ) {
    err=0.0;
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
        iter++;
    }
}
Parallel loop directive

Programmer identifies a block of code as having parallelism, compiler generates a parallel **kernel** for that loop.

```c
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
    y[i] = a*x[i]+y[i];
}
```

**Kernel:** A function that runs in parallel on the **GPU**
while ( err > tol && iter < iter_max ) {
  err=0.0;

  #pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

  #pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
Kernels directive

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```c
#pragma acc kernels
{
  for(int i=0; i<N; i++)
  {
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
  }

  for(int i=0; i<N; i++)
  {
    a(i) = b(i) + c(i)
  }
}
```

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc kernels
    {
        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {

                Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]);

                err = max(err, abs(Anew[j][i] - A[j][i]));
            }
        }

        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }

    iter++;
}
Building the code (using PGI compilers)

$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c

main:

56, Accelerator kernel generated
57, #pragma acc loop gang /* blockIdx.x */
59, #pragma acc loop vector(256) /* threadIdx.x */
56, Generating present_or_copyout(Anew[1:4094][1:4094])
Generating present_or_copyin(A[0:][0:])
Generating NVIDIA Code
Generating compute capability 3.0 binary
59, Loop is parallelizable
63, Max reduction generated for error
68, Accelerator kernel generated
69, #pragma acc loop gang /* blockIdx.x */
71, #pragma acc loop vector(256) /* threadIdx.x */
68, Generating present_or_copyin(Anew[1:4094][1:4094])
Generating present_or_copyout(A[1:4094][1:4094])
Generating NVIDIA Code
Generating compute capability 3.0 binary
71, Loop is parallelizable
Excessive data transfers

while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] – A[j][i]));
        }
    }
}

A, Anew resident on host
A, Anew resident on host
These copies happen every iteration of the outer while loop!*

A, Anew resident on accelerator
A, Anew resident on accelerator

And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
Defining data regions

- The `data` construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
#pragma acc data
{
#pragma acc parallel loop
...
#pragma acc parallel loop
...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.
Data clauses

- **copy ( list )**: Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

- **copyin ( list )**: Allocates memory on GPU and copies data from host to GPU when entering region.

- **copyout ( list )**: Allocates memory on GPU and copies data to the host when exiting region.

- **create ( list )**: Allocates memory on GPU but does not copy.

- **present ( list )**: Data is already present on GPU from another containing data region.

and **present_or_copy[in|out]**, **present_or_create**, **deviceptr**.
Array shaping

• Compiler sometimes cannot determine size of arrays
  – Must specify explicitly using data clauses and array “shape”

C:

```c
#pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
```

Fortran:

```fortran
!$acc data copyin(a(1:end)), copyout(b(s/4:3*s/4))
```

• Note: data clauses can also be used on data, parallel, or kernels e.g.: #pragma acc parallel loop copyin(a)
Jacobi iteration: OpenACC C code (parallel loop + data)

```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Copy A to/from the accelerator only when needed. Create Anew as a device temporary.

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction.Compiler_Directives.pdf
QUESTIONS?

https://connect.iu.edu/ptune15
Class evaluation

- we appreciate your opinion and feedback
- please consider filling-in our class evaluation questionnaire

https://www.surveymonkey.com/r/ptune15