GPU Programming Concepts

Application Porting

Prof. Dr. Matthias S. Müller
Dr. Christian Terboven
Dr. Sandra Wienke
Julian Miller

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

What is This Chapter About?

- Important concepts of programming GPUs
  - Design cycle
  - Offloading concept
  - Creating parallelism
  - Memory management

Design Cycle

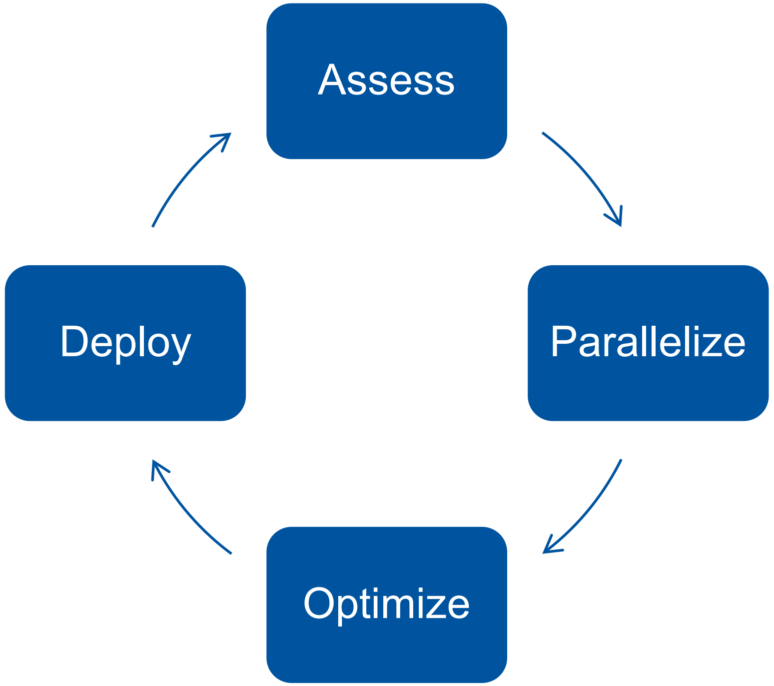
→ Locate hotspots (use a profiler)

→ Evaluate potential for parallelization and GPU acceleration

→ Determine perf. Limits (set goals and expectations)

→ Compare outcome with expectations set in step 1

→ Verify results!



→ If possible: Replace serial hotspot by library call

→ Else if possible: Add directives to the hotspot

→ Else: Write low-level CUDA C/C++, Fortran… code

→ Optimize code iteratively based on the assessment (step 1)

→ Beginning from coarse to fine-grained optimizations

GPGPU Programming

Application

Libraries

"Drop-in" acceleration
- Leave GPU optimization to experts
- Little knowledge and effort needed

cuDNN
cuBLAS
cuSPARSE
cuFFT

Directives

High-level programming
- Leave parallelization and optimization to compilers
- Target a variety of platforms

OpenMP
OpenACC

Programming Languages

Low-level programming
- Maximum flexibility
- Much knowledge and effort required

CUDA: C/C++, Fortran, Python, Java
MATLAB, Mathematica, LabVIEW
OpenCL

GPGPU Programming

- CUDA (Compute Unified Device Architecture)
  - C/C++ (NVIDIA): architecture + programming language, NVIDIA GPUs
  - Fortran (PGI): NVIDIA‘s CUDA for Fortran, NVIDIA GPUs
- OpenCL
  - C (Khronos Group): open standard, portable, CPU/GPU/…
- OpenACC
  - C/Fortran (PGI, Cray, CAPS, NVIDIA): Directive-based accelerator programming for NVIDIA GPUs, published in November 2011
- OpenMP
  - C/C++, Fortran: Directive-based programming for hosts and accelerators, standard, portable, published in July 2013, implementations for Xeon Phis
- …

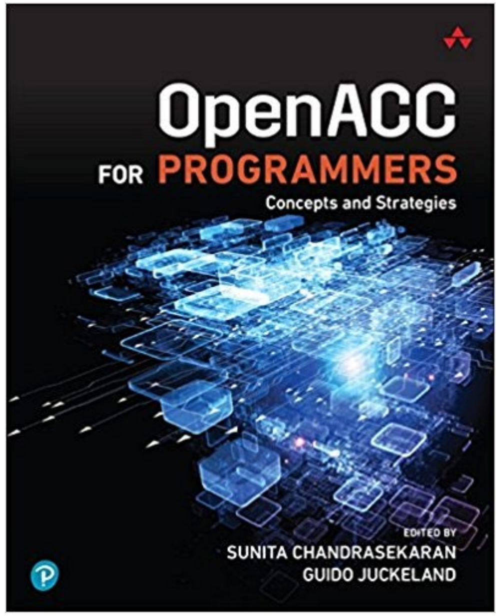
Overview *CUDA* (Compute Unified Device Architecture)

- CUDA C/C++ from NVIDIA
  - Based on industry standard C/C++ (extensions & restrictions)
  - Driver API (low level), Runtime API (higher level)

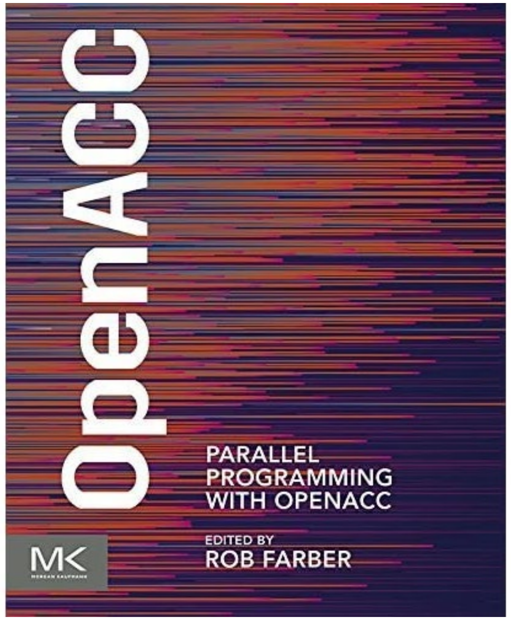
- Brief timeline
  - 2006: Introduction of CUDA, G80 GPU architecture
  - 2007: CUDA Toolkit 1.0
  - 2008: GT200 GPU architecture
  - 2010: Fermi GPU architecture
  - 2012: Kepler K20 GPU architecture
  - 2015: Maxwell GPU architecture
  - 2016: Pascal GPU architecture
  - 2017: Volta GPU architecture
  - 2020: Ampere GPU architecture

Overview *OpenACC*

- Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)
- Today’s members and supporter organizations:
  - Industry: AMD, Cray, NVIDIA, Total,…
  - Universities/Labs: HZDR, EPCC, Virginia Tech, KAUST, Indiana Uni, ORNL, Supercomputing Center Wuxi,…
- Support
  - C,C++ and Fortran
  - NVIDIA GPUs, (AMD GPUs), x86 CPU, OpenPOWER, Sunway, PEZY-SC



OpenACC for Programmers: Concepts and Strategies



Parallel Programming with OpenACC

Nov’11	Spec 1.0
Jun’13	Spec 2.0
Nov’15	Spec 2.5
Nov’16	TR deep copy attach/ detach

Nov’17	Spec 2.6
Nov’18	Spec 2.7
Nov’19	Spec 3.0
Nov’20	Spec 3.1

Overview *OpenMP*

- OpenMP = de-facto standard for shared-memory parallelization
  - From 1997 until now
- Since 2009: OpenMP Accelerator subcommittee
  - Sub group wanted faster development: OpenACC
  - Idea: include lessons learnt into OpenMP standard
- Offloading support since version 4.0 (2013)
- Extended offloading support in 4.5 (2015)
- Full support for accelerator devices in 5.0 (2018)
- Further improvements in accelerator device interactions in 5.1 (2020)

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
Main topics:
- Thread affinity
- Tasking
- Tool support
- Accelerator support

Choice of Programming Model

- Nowadays: GPU APIs (like CUDA, OpenCL) often used
  - More difficult to program but more control
  - Verbose/ may complicate software design and portability

- Directive-based programming model (OpenACC or OpenMP device constructs) delegates responsibility for low-level GPU programming tasks to compiler
  - Data movement
  - Kernel execution
  - Leveraging device-specific features
  - Scheduling work
  - …

→ Many tasks can be done by compiler / runtime
→ User-directed programming
→ Quality of the generated code highly depends on compiler / runtime

→ Choice depends on required control, compiler and library support, and performance target

→ Higher abstraction can reduce development efforts and ease porting

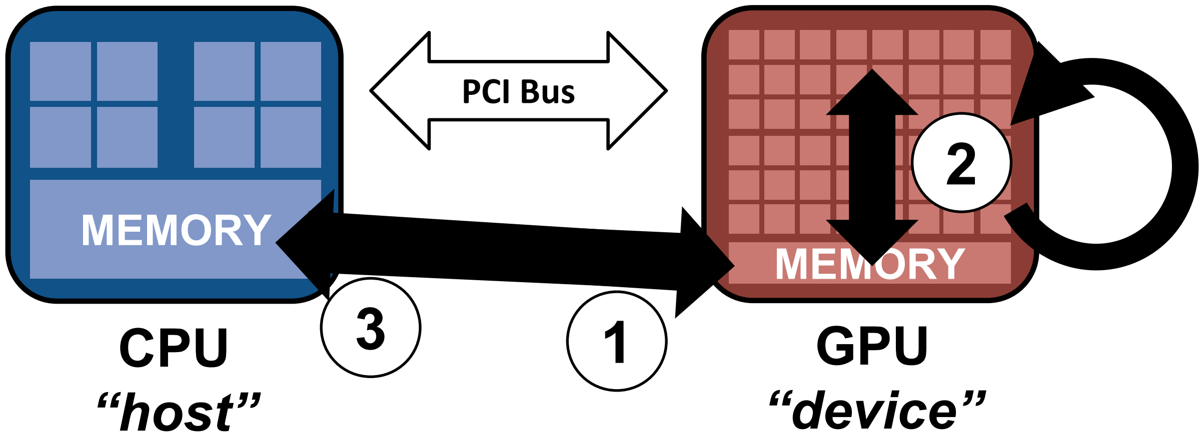
Example DAXPY – CPU

```
void daxpy(int n, double a, double *x, double *y) {
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));
  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  // Invoke daxpy kernel
  daxpy(n, a, x, y);
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

DAXPY = Double-precision real Alpha X Plus Y

$$y = \alpha \cdot x + y$$

Offloading

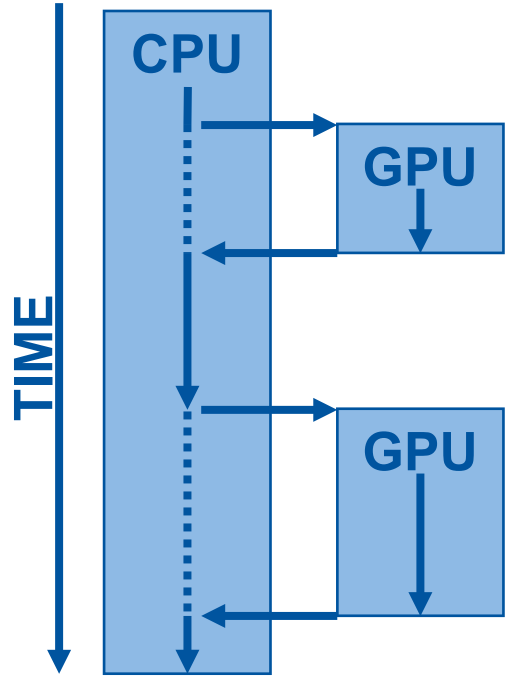


We refer to "discrete GPUs" here.

- Separate host and device memory
  - No coherence between host + device
    - Data transfers needed

- Host-directed execution model
  - Copy input data from CPU mem. to device mem.
  - Execute the device program
  - Copy results from device mem. to CPU mem.

processing flow *(simplified)*



Example DAXPY – GPU

```
void daxpy(int n, double a, double *x,
           double *y)
{
  // 2. Distribute work over GPU
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000;
  static double a = 2.0;

  double *x, *y;
  x = (double *) malloc(n * sizeof(double));
  y = (double *) malloc(n * sizeof(double));

  // Initialize x, y on CPU
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }

  // 1. Allocate data (x, y) on GPU with its
  //    own set of pointers, e.g., d_x, d_y
  // 1. Transfer data (x, y) from CPU to GPU

  // 2. Launch kernel on GPU
  daxpy(n, a, x, y);

  // 3. Transfer result (y) from GPU to CPU

  // Check if all values are 4.0 on CPU

  // 3. Free data (d_x, d_y) on GPU
  free(x); free(y);

  return 0;
}
```