

# OpenACC Tutorial

GridKa School 2017: make science && run

Andreas Herten, Forschungszentrum Jülich, 31 August 2017

## The GPU Platform

- Introduction
- Threading Model
- App Showcase
- Parallel Models

## OpenACC

- History
- OpenMP
- Modus Operandi
- OpenACC's Models

## OpenACC by Example

- OpenACC Workflow
- Identify Parallelism
- Parallelize Loops

- `parallel`
- `loops`
- `pgprof`
- Directive: Kernels

## Data Transfers

- GPU Memory Spaces
- Portability
- Clause: `copy`
- Visual Profiler

## Data Locality

- Analyse Flow
- Directive: `data`

## Optimize

- Levels of Parallelism
- Clause: `gang`
- Memory Coalescing
- Pinned

## Interoperability

- The Keyword
- Tasks

- Task 1
- Task 2
- Task 3
- Task 4

## Conclusions

- List of Tasks

# The GPU Platform

# CPU vs. GPU

*A matter of specialties*



Graphics: Lee [1] and Shearings Holidays [2]

# CPU vs. GPU

*A matter of specialties*



Transporting one

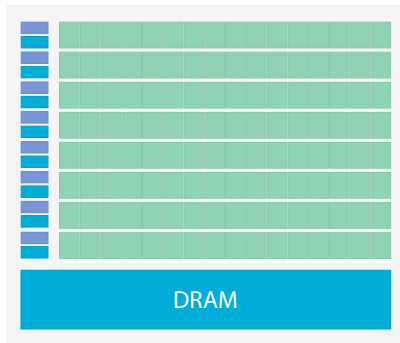
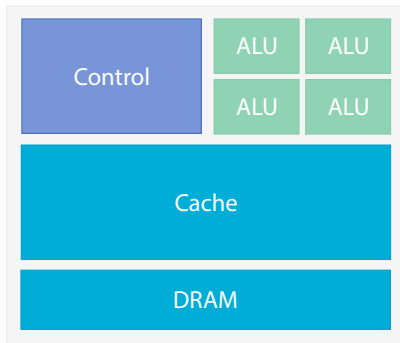


Transporting many

Graphics: Lee [1] and Shearings Holidays [2]

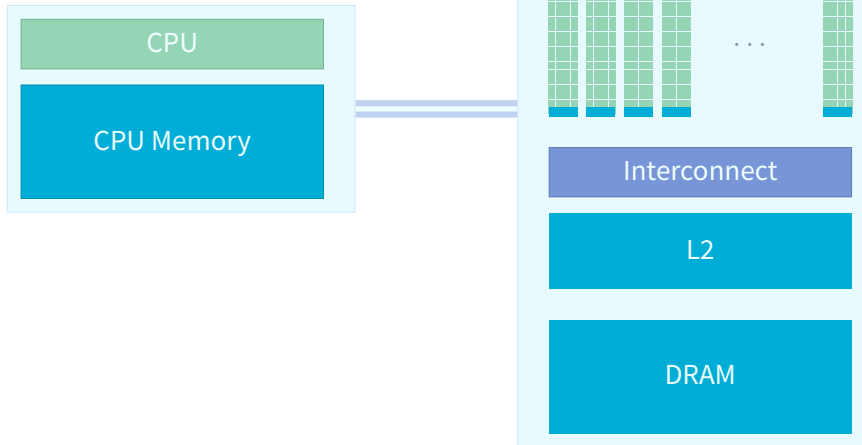
# CPU vs. GPU

*Chip*



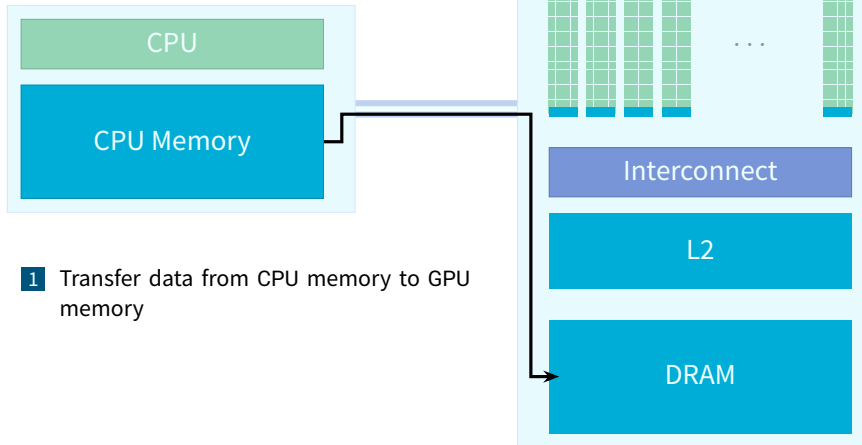
# Processing Flow

*CPU → GPU → CPU*



# Processing Flow

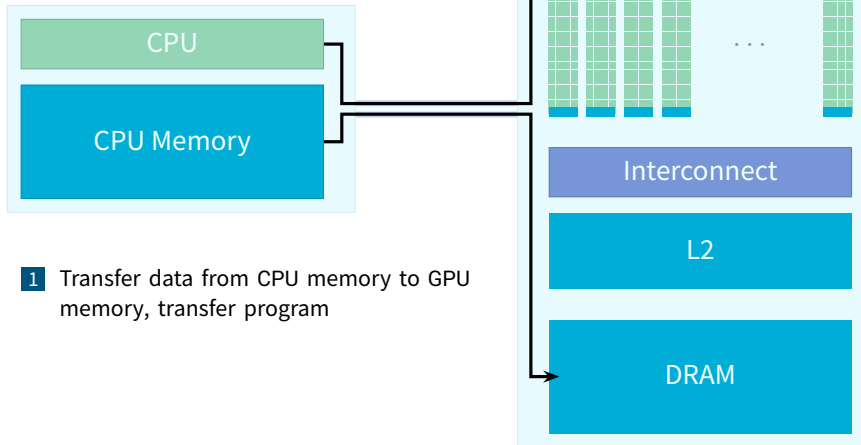
*CPU → GPU → CPU*





# Processing Flow

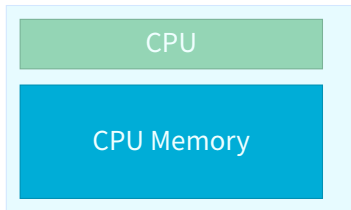
*CPU → GPU → CPU*



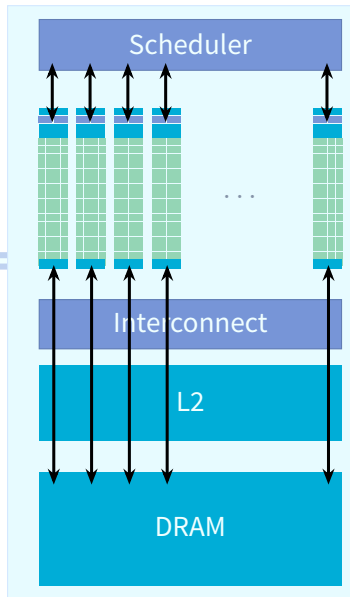
- 1 Transfer data from CPU memory to GPU memory, transfer program

# Processing Flow

*CPU → GPU → CPU*

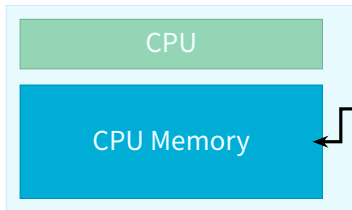


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back

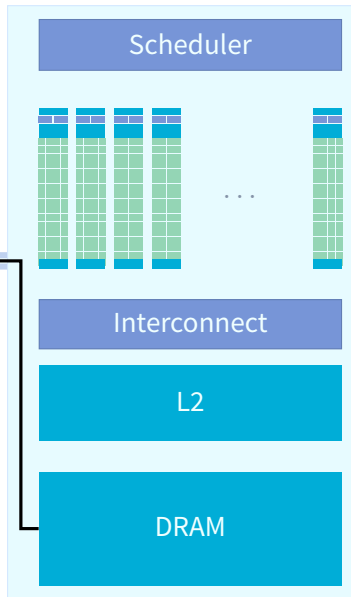


# Processing Flow

*CPU → GPU → CPU*

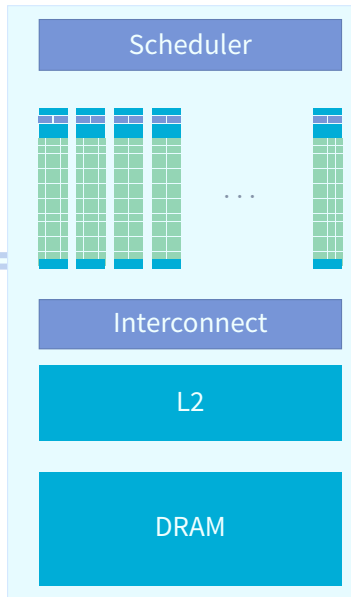
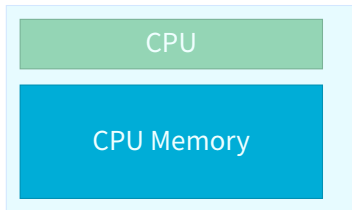


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back
- 3 Transfer results back to host memory



# Processing Flow

*CPU → GPU → CPU*



- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back
- 3 Transfer results back to host memory
  - Old: Manual data transfer invocations – UVA
  - New: Driver automatically transfers data – UM

# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

— Thread  




# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

— Threads  
~~~~~>



# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

— Threads → Block





# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

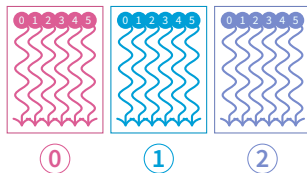
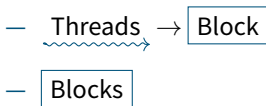
- Threads → Block
- Block



# CUDA Threading Model

*Warp the kernel, it's a thread!*

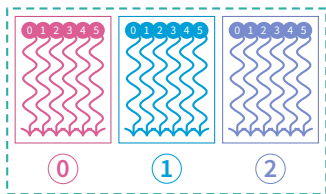
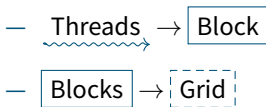
- Methods to exploit parallelism:



# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

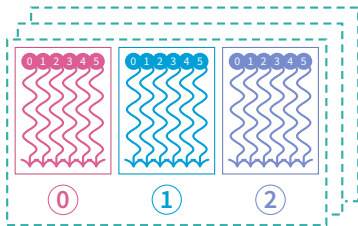


# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- Threads & blocks in 3D



# CUDA Threading Model

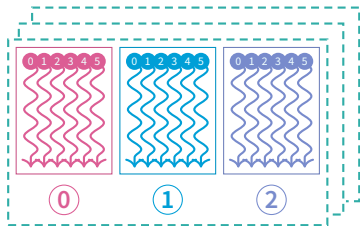
*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- Threads & blocks in 3D

- Threads:** parallel execution units

- Lightweight → fast switching!
- 1000s threads execute simultaneously



# CUDA Threading Model

*Warp the kernel, it's a thread!*

- Methods to exploit parallelism:

- Threads → Block

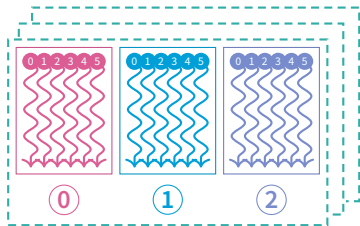
- Blocks → Grid

- Threads & blocks in 3D

- Threads:** parallel execution units

- Lightweight → fast switching!
- 1000s threads execute simultaneously

- Parallel execution unit: **kernel**



### Task 0\*: Setup

- Login to JURON

```
ssh -i mykey train0XX@juron.fz-juelich.de
```

- Directory of tasks

```
cd $HOME/GPU/Tasks/Tasks/
```

- Solutions are always given! You decide when to look.

Directory of solutions: `$HOME/GPU/Tasks/Solutions/`

- Load required modules

```
module load pgi [cuda]
```

- vim is available as editor (or copy files with scp or rsync)

## Task 0: Getting Started

- Change to GPU/Tasks/Task0/ directory
- Read Instructions.rst



Dot Product

GEMM

## Task 0: Getting Started

- Change to GPU/Tasks/Task0/ directory
- Read Instructions.rst

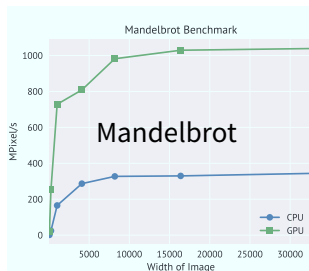
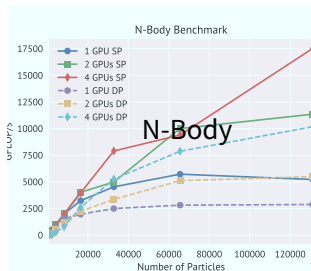
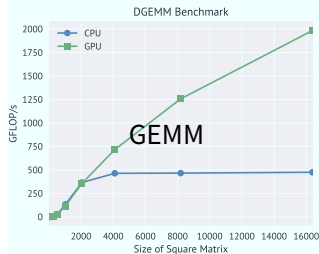
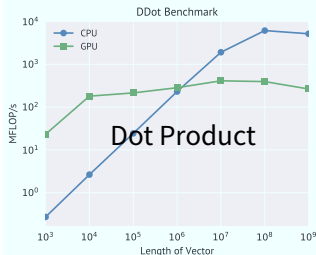
N-Body

Mandelbrot

# Getting GPU-Acquainted

## Some Applications

### TASK 0



# Primer on Parallel Scaling

*Amdahl's Law*

Possible maximum speedup for  $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

# Primer on Parallel Scaling

## *Amdahl's Law*

Possible maximum speedup for  $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

**$N$  Processors**  $t(N) = t_s + t_p/N$

# Primer on Parallel Scaling

*Amdahl's Law*

Possible maximum speedup for  $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

**$N$  Processors**  $t(N) = t_s + t_p/N$

**Speedup**  $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p/N}$

**Efficiency:**  $\varepsilon = s/N$

# Primer on Parallel Scaling

*Amdahl's Law*

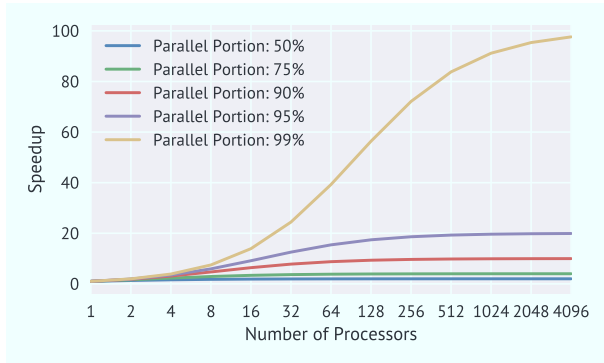
Possible maximum speedup for  $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

**$N$  Processors**  $t(N) = t_s + t_p/N$

**Speedup**  $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p/N}$

**Efficiency:**  $\varepsilon = s/N$

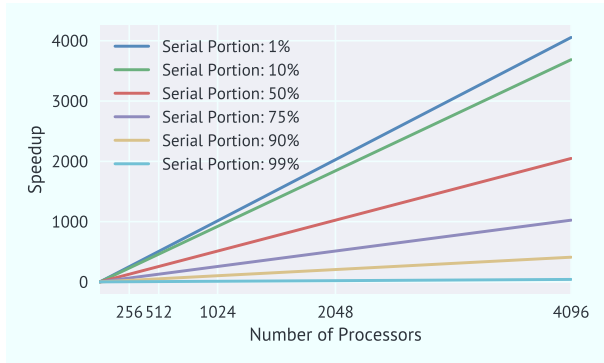


# Primer on Parallel Scaling II

## Gustafson-Barsis's Law

*[...] speedup should be measured by scaling the problem to the number of processors, not fixing problem size.*

*– John Gustafson*



Parallel programming is not easy!

Things to consider:

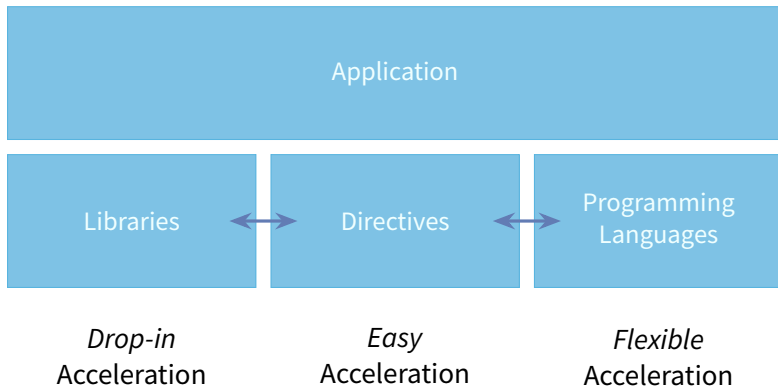
- Is my application **computationally intensive** *enough*?
- What are the levels of **parallelism**?
- How much **data** needs to be **transferred**?
- Is the **gain** worth the **pain**?



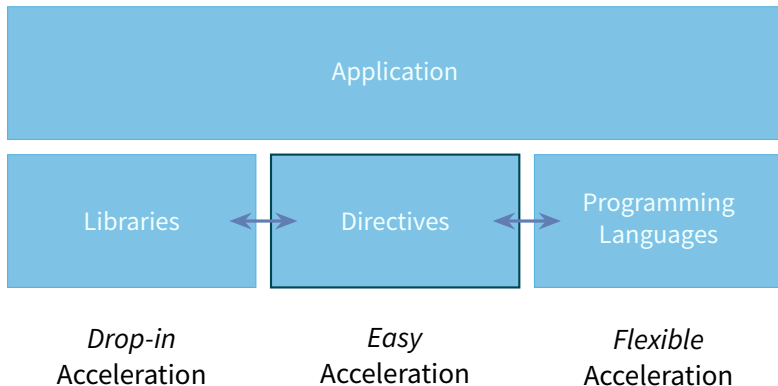
Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*...

- OpenACC
- OpenMP
- Thrust
- PyCUDA
- CUDA Fortran
- CUDA
- OpenCL

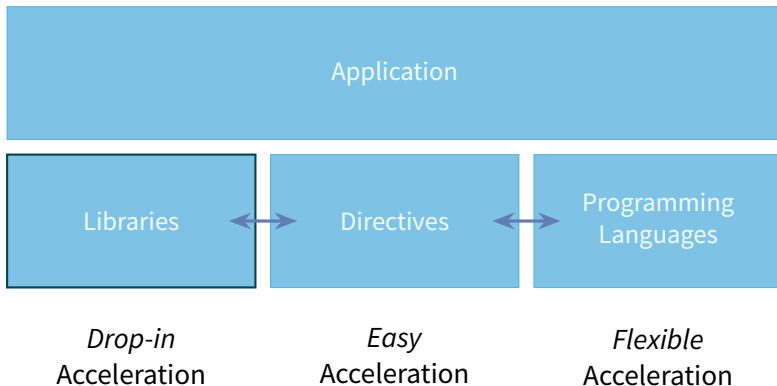
# Summary of Acceleration Possibilities



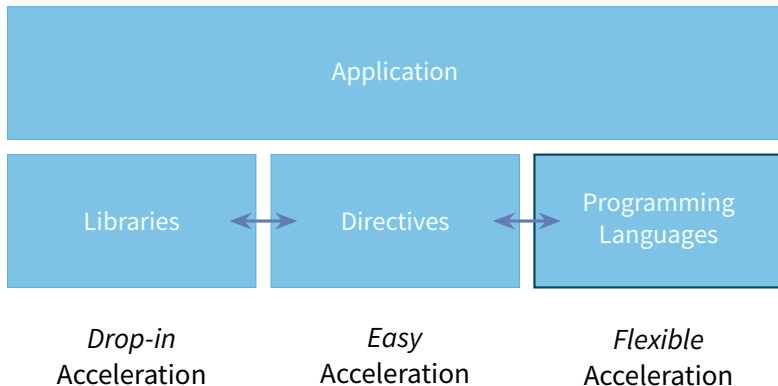
# Summary of Acceleration Possibilities



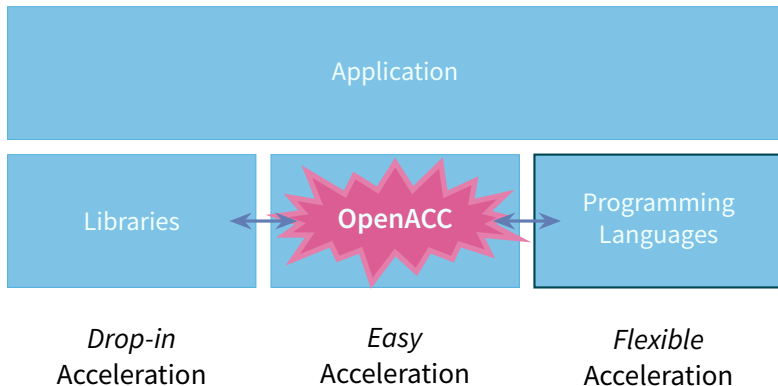
# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



2011 OpenACC 1.0 specification is released 

*NVIDIA, Cray, PGI, CAPS*

2013 OpenACC 2.0: More functionality, portability 

2015 OpenACC 2.5: Enhancements, clarifications 

2016 OpenACC 2.6 proposed (deep copy, ...) 

→ <https://www.openacc.org/>

Also: *Best practice guide* 

- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

*But OpenMP 4.0 now also has offloading feature*

- Fork/join model

*Master thread launches parallel child threads; merge after execution*

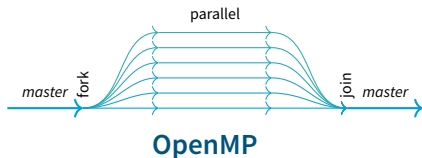


- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

*But OpenMP 4.0 now also has offloading feature*

- Fork/join model

*Master thread launches parallel child threads; merge after execution*

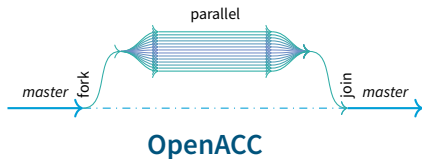
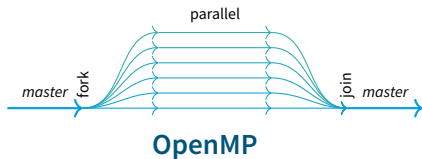


- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

*But OpenMP 4.0 now also has offloading feature*

- Fork/join model

*Master thread launches parallel child threads; merge after execution*



- 1 Annotate code with directives, indicating parallelism
- 2 OpenACC-capable compiler generates accelerator-specific code
- 3 \$success

# 1 Directives

*pragmatic*

- Compiler directives state intend to compiler

**C/C++**

```
#pragma acc kernels  
for (int i = 0; i < 23; i++)  
// ...
```

**Fortran**

```
!$acc kernels  
do i = 1, 24  
! ...  
!$acc end kernels
```

- Ignored by compiler which does not understand OpenACC
- High level programming model for accelerators; heterogeneous programs
- OpenACC: Compiler directives, library routines, environment variables
- Portable across host systems and accelerator architectures

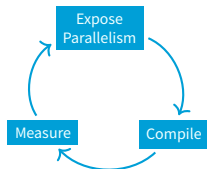
## 2 Compiler

*Simple and abstracted*

- Compiler support
  - PGI *Best performance, great support, free*
  - GCC *Beta, limited coverage, OSS*
  - Cray ???
- Trust compiler to generate intended parallelism; check status output!
- No need to know ins'n'outs of accelerator; leave it to expert compiler engineers
- One code can target different accelerators: GPUs, or even multi-core CPUs → **Portability**

### 3 \$uccess

*Iteration is key*

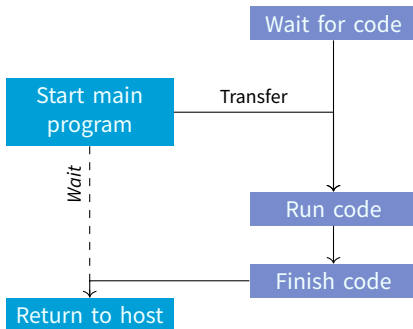


- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple → refine

#### ⇒ **Productivity**

- Because of *generalness*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, ...)

- Main program executes on **host**
- Device code is transferred to **accelerator**
- Execution on accelerator is started
- Host waits until return (except: async)





- Usually: Two separate memory spaces
- Data needs to be transferred to device for computation; needs to be transferred back for further evaluation
  - Transfers hidden from programmer – **caution**: latency, bandwidth, memory size
  - Memories are not coherent
  - Compiler helps; GPU runtime helps



- OpenACC interpretation needs to be activated as compile flag

**PGI** `pgcc -acc [-ta=tesla]`

**GCC** `gcc -fopenacc`

- Additional flags possible to improve/modify compilation

`-ta=tesla:cc60` Use compute capability 6.0

`-ta=tesla:lineinfo` Add source code correlation into binary

`-ta=tesla:managed` Use unified memory

`-fopenacc-dim=geom` Use *geom* configuration for threads

- Compiler directives, ignored by incapable compilers
- Similar to OpenMP
- Support for GPU, multicore CPU, other accelerators (Intel Xeon Phi)
- Syntax **C/C++**  
`#pragma acc directive [clause, [, clause] ...] newline`
- Syntax **Fortran**  
`!$acc directive [clause, [, clause] ...]`  
`!$acc end directive`

```
#pragma acc data copy(x[0:N],y[0:N])  
#pragma acc parallel loop  
{  
    for (int i=0; i<N; i++) {  
        x[i] = 1.0;  
        y[i] = 2.0;  
    }  
    for (int i=0; i<N; i++) {  
        y[i] = i*x[i]+y[i];  
    }  
}
```

# OpenACC by Example

Identify available parallelism



Parallelize loops with OpenACC

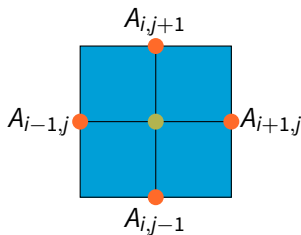


Optimize data locality



Optimize loop performance

- Example for acceleration: **Jacobi solver**
- Iterative solver, converges to correct value
- Each iteration step: compute average of neighboring points
- Example: 2D Poisson equation:  $\nabla^2 A(x, y) = B(x, y)$



$$A_{k+1}(i, j) = -\frac{1}{4} (B(i, j) - (A_k(i-1, j) + A_k(i, j+1) + A_k(i+1, j) + A_k(i, j-1)))$$

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error,
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
        }
    }
    for (int iy = iy_start; iy < iy_end; iy++) {
        for (int ix = ix_start; ix < ix_end; ix++) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
        }
    }
    for (int ix = ix_start; ix < ix_end; ix++) {
        A[0*nx+ix] = A[(ny-2)*nx+ix];
        A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
    }  
    for (int iy = iy_start; iy < iy_end; iy++) {  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[iy*nx+ix] = Anew[iy*nx+ix];  
        }  
    }  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        A[0*nx+ix] = A[(ny-2)*nx+ix];  
        A[(ny-1)*nx+ix] = A[1*nx+ix];  
    }  
    // same for iy  
    iter++;  
}
```

Iterate until converged



```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
    }  
    for (int iy = iy_start; iy < iy_end; iy++) {  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[iy*nx+ix] = Anew[iy*nx+ix];  
        }  
    }  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        A[0*nx+ix] = A[(ny-2)*nx+ix];  
        A[(ny-1)*nx+ix] = A[1*nx+ix];  
    }  
    // same for iy  
    iter++;  
}
```

Iterate until converged

Iterate across  
matrix elements

```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
    }  
    for (int iy = iy_start; iy < iy_end; iy++) {  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[iy*nx+ix] = Anew[iy*nx+ix];  
        }  
    }  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        A[0*nx+ix] = A[(ny-2)*nx+ix];  
        A[(ny-1)*nx+ix] = A[1*nx+ix];  
    }  
    // same for iy  
    iter++;  
}
```

Iterate until converged

Iterate across  
matrix elements

Calculate new value  
from neighbors

```

while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error,
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
        }
    }
    for (int iy = iy_start; iy < iy_end; iy++) {
        for (int ix = ix_start; ix < ix_end; ix++) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
        }
    }
    for (int ix = ix_start; ix < ix_end; ix++) {
        A[0*nx+ix] = A[(ny-2)*nx+ix];
        A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}

```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Accumulate error

```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            for (int ix = ix_start; ix < ix_end; ix++) {  
                A[iy*nx+ix] = Anew[iy*nx+ix];  
            }  
        }  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[0*nx+ix] = A[(ny-2)*nx+ix];  
            A[(ny-1)*nx+ix] = A[1*nx+ix];  
        }  
        // same for iy  
        iter++;  
    }  
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Accumulate error

Swap input/output

```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            for (int ix = ix_start; ix < ix_end; ix++) {  
                A[iy*nx+ix] = Anew[iy*nx+ix];  
            }  
        }  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[0*nx+ix] = A[(ny-2)*nx+ix];  
            A[(ny-1)*nx+ix] = A[1*nx+ix];  
        }  
        // same for iy  
        iter++;  
    }  
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Accumulate error

Swap input/output

Set boundary conditions

**Identify available parallelism**



Parallelize loops with OpenACC



Optimize data locality



Optimize loop performance

- Use pgprof to analyze unaccelerated version of Jacobi solver
- Investigate!

## Task 1: Analyze Application

- Change to Task1/ directory
- Compile: `make task1`  
*Usually, compile just with make (but this exercise is special)*
- Submit *profiling run* to the batch system:  
`make task1_profile`  
*Study bsub call and pgprof call; try to understand*

- Use pgprof to analyze unaccelerated version of Jacobi solver
- Investigate!

## Task 1: Analyze Application

- Change to Task1/ directory
- Compile: `make task1`  
*Usually, compile just with make (but this exercise is special)*
- Submit *profiling run* to the batch system:  
`make task1_profile`  
*Study bsub call and pgprof call; try to understand*

??? **Where is hotspot? Which parts should be accelerated?**



# Profile of Application

*Info during compilation*

```
$ pgcc -DUSE_DOUBLE -Minfo=all,intensity -fast -Minfo=ccff -Mprof=ccff  
poisson2d_reference.o poisson2d.c -o poisson2d  
poisson2d.c:  
main:  
    68, Generated vector simd code for the loop  
        FMA (fused multiply-add) instruction(s) generated  
    98, FMA (fused multiply-add) instruction(s) generated  
   105, Loop not vectorized: data dependency  
   123, Loop not fused: different loop trip count  
        Loop not vectorized: data dependency  
        Loop unrolled 8 times
```

- Automated optimization of compiler, due to -fast
- Vectorization, FMA, unrolling

# Profile of Application

*Info during run*

```
=====  
===== CPU profiling result (flat):  
Time(%)      Time   Name  
77.52%    999.99ms  main (poisson2d.c:148 0x6d8)  
 9.30%     120ms   main (0x704)  
 7.75%    99.999ms  main (0x718)  
 0.78%    9.9999ms  main (poisson2d.c:128 0x348)  
 0.78%    9.9999ms  main (poisson2d.c:123 0x398)  
 0.78%    9.9999ms  __xlmass_expd2 (0xffcc011c)  
 0.78%    9.9999ms  __c_mcopy8 (0xffcc0054)  
 0.78%    9.9999ms  __xlmass_expd2 (0xffcc0034)  
===== Data collected at 100Hz frequency
```

- 78 % in main()
- Since everything is in main – limited helpfulness
- Let's look into main!

# Code Interdependency Analysis

*What is independent?*

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error,
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
        }
    }
    for (int iy = iy_start; iy < iy_end; iy++) {
        for (int ix = ix_start; ix < ix_end; ix++) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
        }
    }
    for (int ix = ix_start; ix < ix_end; ix++) {
        A[0*nx+ix] = A[(ny-2)*nx+ix];
        A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

# Code Dependency Analysis

*What is independent?*

```
while ( error > tol && iter < iter_max ) {  
    error = 0.0;  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        for (int iy = iy_start; iy < iy_end; iy++) {  
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -  
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]  
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));  
            error = fmaxr(error,  
                ↪ fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));  
        }  
    }  
    for (int iy = iy_start; iy < iy_end; iy++) {  
        for (int ix = ix_start; ix < ix_end; ix++) {  
            A[iy*nx+ix] = Anew[iy*nx+ix];  
        }  
    }  
    for (int ix = ix_start; ix < ix_end; ix++) {  
        A[0*nx+ix] = A[(ny-2)*nx+ix];  
        A[(ny-1)*nx+ix] = A[1*nx+ix];  
    }  
    // same for iy  
    iter++;  
}
```

Data dependency  
between iterations

Independent loop  
iterations

Independent loop  
iterations

Independent loop  
iterations

Identify available parallelism



**Parallelize loops with OpenACC**



Optimize data locality



Optimize loop performance

# Parallel Loops: Parallel

*Maybe the second most important directive*

- Programmer identifies block containing parallelism → compiler generates GPU code (*kernel*)
- Program launch creates *gangs* of parallel threads on GPU
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

## 🔗 OpenACC: parallel

```
#pragma acc parallel [clause, [, clause] ...] newline  
{structured block}
```

Diverse clauses to augment the parallel region

`private(var)` A copy of variables `var` is made for each gang

`firstprivate(var)` Same as `private`, except `var` will be initialized with value from host

`if(cond)` Parallel region will execute on accelerator only if `cond` is true

`reduction(op:var)` Reduction is performed on variable `var` with operation `op`; supported: `+` `*` `max` `min` ...

`async[(int)]` No implicit barrier at end of parallel region

# Parallel Loops: Loops

*Maybe the third most important directive*

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Optional: Describe type of parallelism

## 🔧 OpenACC: loop

```
#pragma acc loop [clause, [, clause] ...] newline  
{structured block}
```



- `independent` Iterations of loop are data-independent (implied if in `parallel` region (and no `seq` or `auto`))
- `collapse(int)` Collapse `int` tightly-nested loops
  - `seq` This loop is to be executed sequentially (not parallel)
- `tile(int[,int])` Split loops into loops over tiles of the full size
- `auto` Compiler decides what to do

# Parallel Loops: Parallel Loops

*Maybe the most important directive*

- Combined directive: shortcut  
*Because its used so often*
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

🔗 OpenACC: `parallel loop`

```
#pragma acc parallel loop [clause, [, clause] ...]
```

# Parallel Loops Example

```
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}
#pragma acc parallel loop reduction(+:sum)
{
    for (int i=0; i<N; i++) {
        y[i] = i*x[i]+y[i];
        sum+=y[i];
    }
}
```

Kernel 1

Kernel 2

- Add OpenACC parallelism to main loop in Jacobi
  - Profile code
- Congratulations, you are a GPU developer!

## Task 2: A First Parallel Loop

- Change to Task2/ directory
- Compile: `make`
- Submit parallel run to the batch system: `make run`  
*Adapt the `bsub` call and run with other number of iterations, matrix sizes*
- Profile: `make profile`  
*`pgprof` or `nvprof` is prefix to call to `poisson2d`*

```
$ make
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60,managed
  poisson2d_reference.c -o poisson2d_reference.o
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60,managed poisson2d.c
  poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    109, Accelerator kernel generated
        Generating Tesla code
    109, Generating reduction(max:error)
    110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    112, #pragma acc loop seq
    109, Generating implicit copyin(A[:,rhs[:]])
        Generating implicit copyout(Anew[:])
    112, Complex loop carried dependence of Anew-> prevents parallelization
        Loop carried dependence of Anew-> prevents parallelization
        Loop carried backward dependence of Anew-> prevents vectorization
```

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4444> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 60.0827 s, This: 9.5541 s, speedup: 6.29
```

- Profiles applications, mainly for NVIDIA GPUs, but also CPU code
  - GPU: CUDA kernels, API calls, OpenACC
  - pgprof vs nvprof: Twins with other configurations
  - Generate concise performance reports, full timelines; measure events and metrics (hardware counters)
- ⇒ Powerful tool for GPU application analysis
- <http://docs.nvidia.com/cuda/profiler-users-guide/>

# Profile of Jacobi

With pgprof

```
$ make profile
==116606== PGPROF is profiling process 116606, command: ./poisson2d 10
==116606== Profiling application: ./poisson2d 10
Jacobi relaxation calculation: max 10 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
2048x2048: Ref: 0.8378 s, This: 0.2716 s, speedup: 3.08
==116606== Profiling result:
Time(%)   Time      Calls      Avg      Min      Max      Name
99.96%  129.82ms      10  12.982ms  11.204ms  20.086ms  main_109_gpu
0.02%   30.560us      10   3.0560us  2.6240us  3.8720us  main_109_gpu_red
0.01%   10.304us      10   1.0300us   960ns   1.2480us  [CUDA memcpy HtoD]
0.00%    6.3680us      10    636ns    608ns    672ns    [CUDA memcpy DtoH]

==116606== Unified Memory profiling result:
Device "Tesla P100-SXM2-16GB (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3360  204.80KB  64.000KB  960.00KB  672.0000MB  25.37254ms  Host To Device
    3200  204.80KB  64.000KB  960.00KB  640.0000MB  30.94435ms  Device To Host
     2454      -      -      -      -      66.99111ms  GPU Page fault groups
Total CPU Page faults: 2304

==116606== API calls:
Time(%)   Time      Calls      Avg      Min      Max      Name
58.17%   639.81ms       5   127.96ms   564ns  189.20ms  cuDevicePrimaryCtxRetain
26.35%   289.79ms       4    72.449ms  69.684ms   74.126ms  cuDevicePrimaryCtxRelease
```



# Profile of Jacobi

With pgprof

```
$ make profile
==116606== PGPROF is profiling process 116606, command: ./poisson2d 10
==116606== Profiling application: ./poisson2d 10
Jacobi relaxation calculation: max 10 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
2048x2048: Ref: 0.8378 s, This: 0.2716 s, speedup: 3.08
```

*Only one function is parallelized!*  
*Let's do the rest!*

```
Device "Tesla P100-SXM2-16GB (0)"
```

| Count | Avg Size | Min Size | Max Size | Total Size | Total Time | Name                  |
|-------|----------|----------|----------|------------|------------|-----------------------|
| 3360  | 204.80KB | 64.000KB | 960.00KB | 672.0000MB | 25.37254ms | Host To Device        |
| 3200  | 204.80KB | 64.000KB | 960.00KB | 640.0000MB | 30.94435ms | Device To Host        |
| 2454  | -        | -        | -        | -          | 66.99111ms | GPU Page fault groups |

```
Total CPU Page faults: 2304
```

```
==116606== API calls:
```

| Time(%) | Time     | Calls | Avg      | Min      | Max      | Name                      |
|---------|----------|-------|----------|----------|----------|---------------------------|
| 58.17%  | 639.81ms | 5     | 127.96ms | 564ns    | 189.20ms | cuDevicePrimaryCtxRetain  |
| 26.35%  | 289.79ms | 4     | 72.449ms | 69.684ms | 74.126ms | cuDevicePrimaryCtxRelease |

# More Parallelism: Kernels

*More freedom for compiler*

- Kernels directive: second way to expose parallelism
- Region may contain parallelism
- Compiler determines parallelization opportunities

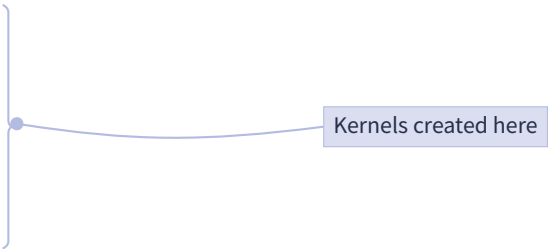
→ More freedom for compiler

- Rest: Same as for parallel

## 🔑 OpenACC: kernels

```
#pragma acc kernels [clause, [, clause] ...] newline  
structured block
```

```
double sum = 0.0;
#pragma acc kernels
{
  for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
  }
  for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
  }
}
```



Kernels created here

- Both approaches equally valid; can perform equally well

- Both approaches equally valid; can perform equally well
- **kernels**
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- **parallel**
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - Similar to OpenMP

- Both approaches equally valid; can perform equally well
- **kernels**
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- **parallel**
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - Similar to OpenMP
- Both regions may not contain other kernels/parallel regions
- No branching into or out
- Program must not depend on order of evaluation of clauses
- At most: One if clause

- Add OpenACC parallelism to other loops of **while** (L:123 – L:141)
- Use either `kernel`s or `parallel`
- Do they perform equally well?

## Task 3: More Parallel Loops

- Change to Task3/ directory
- Change source code
- Compile: `make`  
*Study the compiler output!*
- Submit parallel run to the batch system: `make run`

```
$ make
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60,managed
  poisson2d_reference.c -o poisson2d_reference.o
poisson2d.c:
main:
    109, Accelerator kernel generated
        Generating Tesla code
    109, Generating reduction(max:error)
    110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    112, #pragma acc loop seq
    109, ...
    121, Accelerator kernel generated
        Generating Tesla code
    124, #pragma acc loop gang /* blockIdx.x */
    126, #pragma acc loop vector(128) /* threadIdx.x */
    121, Generating implicit copyin(Anew[:])
        Generating implicit copyout(A[:])
    126, Loop is parallelizable
    133, Accelerator kernel genera...
```



# Parallel Jacobi II

## Run result

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4458> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc15>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref:  64.9401 s, This:  0.4099 s, speedup:  158.45
```

# Parallel Jacobi II

## Run result

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4458> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc15>>
Jacobi relaxation calculation on 2048 x 2048 mesh
Calculate reference solution on serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 64.9401 s, This: 0.4099 s, speedup: 158.45
```

Done?!

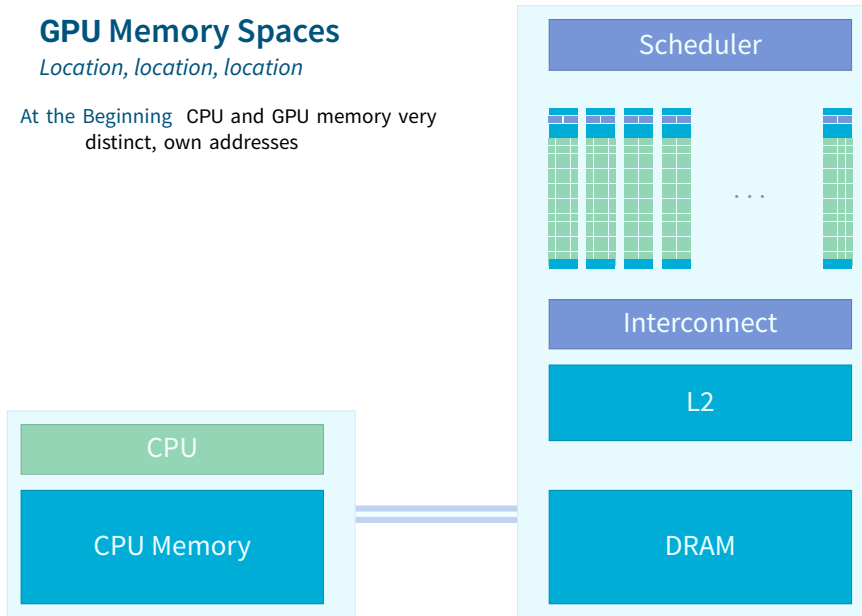
```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc parallel loop reduction(max:error)
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
                  + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
        }
    }
    #pragma acc parallel loop
    for (int iy = iy_start; iy < iy_end; iy++) {
        for (int ix = ix_start; ix < ix_end; ix++) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
        }
    }
    #pragma acc parallel loop
    for (int ix = ix_start; ix < ix_end; ix++) {
        A[0*nx+ix] = A[(ny-2)*nx+ix];
        A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

- Up to now: We did not care about **data transfers**
- Compiler and runtime care
- Magic keyword: `-ta=tesla:managed`
- Only feature of (recent) NVIDIA GPUs!

# GPU Memory Spaces

*Location, location, location*

At the Beginning CPU and GPU memory very distinct, own addresses

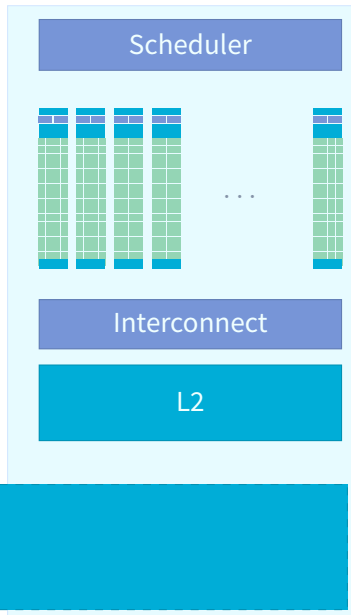


# GPU Memory Spaces

*Location, location, location*

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual



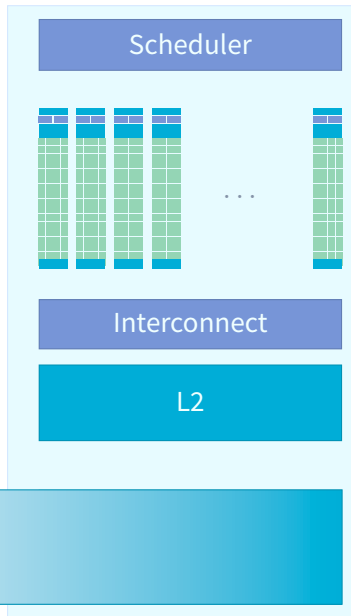
# GPU Memory Spaces

*Location, location, location*

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory\*: Data copy by driver, but whole data at once (Kepler)



# GPU Memory Spaces

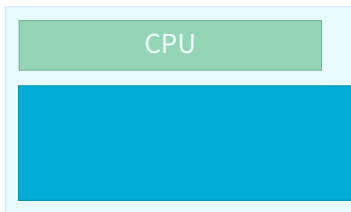
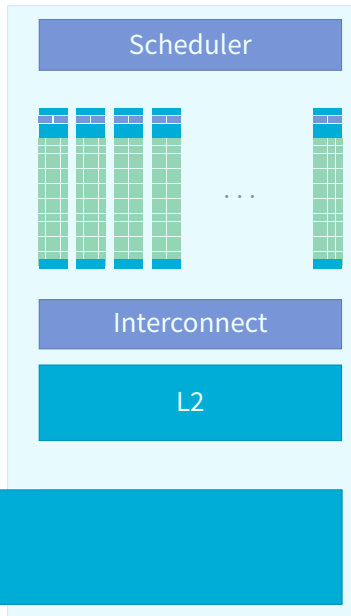
*Location, location, location*

At the **Beginning** CPU and GPU memory very distinct, own addresses

**CUDA 4.0** Unified Virtual Addressing: pointer from same address pool, but data copy manual

**CUDA 6.0** Unified Memory\*: Data copy by driver, but whole data at once (Kepler)

**CUDA 8.0** Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)



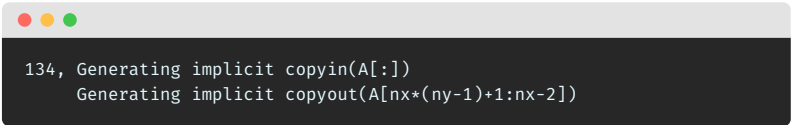


- Managed memory: Only NVIDIA GPU feature
  - Great OpenACC features: Portability
- Code should also be fast without `-ta=tesla:managed!`
- Let's remove it from compile flags!

- Managed memory: Only NVIDIA GPU feature
  - Great OpenACC features: Portability
- Code should also be fast without `-ta=tesla:managed!`
- Let's remove it from compile flags!

```
$ make
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60
poisson2d_reference.c -o poisson2d_reference.o
poisson2d.c:
PGC-S-0155-Compiler failed to translate accelerator region
(see -Minfo messages): Could not find allocated-variable index for
symbol (poisson2d.c: 110)
...
PGC/power Linux 17.4-0: compilation completed with severe errors
```

- Compiler implicitly created copy clauses to copy data to device



```
134, Generating implicit copyin(A[:])  
Generating implicit copyout(A[nx*(ny-1)+1:nx-2])
```

- It couldn't determine length of copied data ...
- ...but before: no problem – Unified Memory!

- Compiler implicitly created copy clauses to copy data to device



```
134, Generating implicit copyin(A[:])  
Generating implicit copyout(A[nx*(ny-1)+1:nx-2])
```

- It couldn't determine length of copied data ...
- ...but before: no problem – Unified Memory!
- Now: Problem!
- We need to give that information! (see also [later](#))

## 🚀 OpenACC: copy

```
#pragma acc parallel copy(A[start:end])
```

```
Also: copyin(B[s:e]) copyout(C[s:e]) present(D[s:e]) create(E[s:e])
```

# Data Copies

*Tell compiler which data is needed where*

## TASK 4

- Add copy clauses to parallel regions
- Profile with Visual Profiler

### Task 4: Data Copies

- Change to Task4/ directory
- Work on TODOs
- Compile: `make`
- Submit parallel run to the batch system: `make run`  
*It might take some time*
- Generate profile with `make profile_tofile`

```
$ make
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60
poisson2d_reference.c -o poisson2d_reference.o
poisson2d.c:
main:
    109, Generating copy(A[:ny*nx],Anew[:ny*nx],rhs[:ny*nx])
        ...
    121, Generating copy(Anew[:ny*nx],A[:ny*nx])
        ...
    131, Generating copy(A[:ny*nx])
        Accelerator kernel generated
        Generating Tesla code
    132, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    137, Generating copy(A[:ny*nx])
        Accelerator kernel generated
        Generating Tesla code
    138, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

# Data Copies

## Run Result

```
$ make run
<<Starting on juronc13>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 114.7186 s, This: 25.0522 s, speedup: 4.58
```

# Data Copies

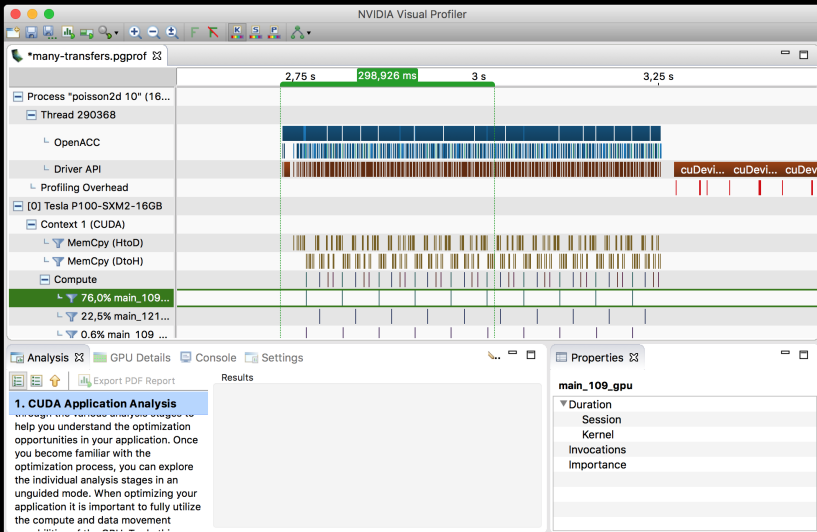
## Run Result

```
$ make run
<<Starting on juronc13>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution on 1 serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 114.7186 s, This: 25.0522 s, speedup: 4.58
```

*Slower?!  
Why?*

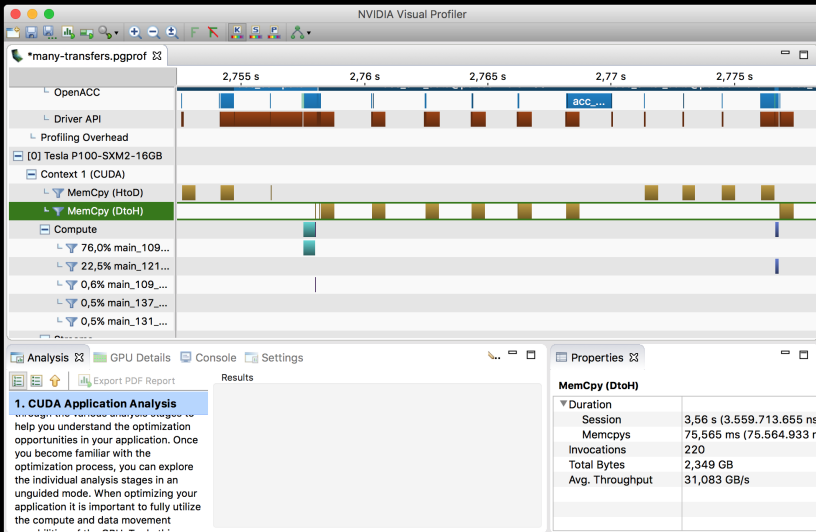


- GUI tool accompanying pgprof / nvprof
    - PGI Start pgprof without parameters
    - NVIDIA Start nvvp
  - Timeline view of all things GPU
    - Study stages and interplay of application
  - Interactive or with input from command line profilers
  - View launch and run configurations
  - Guided and unguided analysis
- <https://developer.nvidia.com/nvidia-visual-profiler>



# Jacobi in Visual Profiler

Zoom into kernel calls



Identify available parallelism



Parallelize loops with OpenACC



**Optimize data locality**



Optimize loop performance

# Analyze Jacobi Data Flow

*In code*

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

*#pragma acc parallel loop*

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

```
    iter++  
}
```

# Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

copy

*#pragma acc parallel loop*

A, Anew resident on device

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

```
    iter++
```

```
}
```

# Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

copy

*#pragma acc parallel loop*

A, Anew resident on device

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

A, Anew resident on device

```
        iter++  
    }  
}
```

# Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

copy

*#pragma acc parallel loop*

A, Anew resident on device

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

A, Anew resident on host

A, Anew resident on device

```
    iter++  
}
```



# Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

copy

*#pragma acc parallel loop*

A, Anew resident on device

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

A, Anew resident on host

A, Anew resident on device

```
    iter++  
}
```



# Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {  
    error = 0.0;
```

A, Anew resident on host

copy

*#pragma acc parallel loop*

A, Anew resident on device

Copies are done  
in each iteration!

```
for (int ix = ix_start; ix <  
    ↪ ix_end; ix++) {  
    for (int iy = iy_start; iy <  
        ↪ iy_end; iy++) {  
        // ...  
    }  
}
```

A, Anew resident on host

A, Anew resident on device

iter++

}



- Meanwhile, whole algorithm is using GPU
- At beginning of **while** loop, data copied to device; at end of loop, copied by to host
- Depending on type of parallel regions in **while** loop: Data copied in between regions as well

- Meanwhile, whole algorithm is using GPU
- At beginning of **while** loop, data copied to device; at end of loop, copied by to host
- Depending on type of parallel regions in **while** loop: Data copied in between regions as well
- **Slow! Data copies are expensive!**

# Data Regions

*To manually specify data locations*

- Defines region of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers

## 🔑 OpenACC: data

```
#pragma acc data [clause, [, clause] ...] newline  
{structured block}
```

### Clauses to augment the data regions

- `copy(var)` Allocates memory of `var` on GPU, copies data to GPU at beginning of region, copies data to host at end of region  
Specifies size of `var`: `var[lowerBound:size]`
- `copyin(var)` Allocates memory of `var` on GPU, copies data to GPU at beginning of region
- `copyout(var)` Allocates memory of `var` on GPU, copies data to host at end of region
- `create(var)` Allocates memory of `var` on GPU
- `present(var)` Data of `var` is not copied automatically to GPU but considered present

```
#pragma acc data copyout(y[0:N]) create(x[0:N])
{
  double sum = 0.0;
  #pragma acc parallel loop
  for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
  }
  #pragma acc parallel loop
  for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
  }
}
```

- Add data region such that all data resides on device during iterations
- Optional: See your success in Visual Profiler

## Task 5: Data Region

- Change to Task5/ directory
- Work on TODOs
- Compile: `make`
- Submit to the batch system: `make run`
- Generate profile with `make profile_tofile`



```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.c
poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    104, Generating copyin(rhs[:ny*nx])
        Generating create(Anew[:ny*nx])
        Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
        Generating Tesla code
    110, Generating reduction(max:error)
    111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    113, #pragma acc loop seq
    ...
```

```
$ make run
<<Starting on juronc12>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 115.0765 s, This:   0.4807 s, speedup:   239.38
```

```
$ make run
```

```
<<Starting on juronc12>>
```

```
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
```

```
Calcul
```

```
0,
```

```
100,
```

```
200,
```

```
Calcul
```

```
0, 0.249999
```

```
100, 0.249760
```

```
200, 0...
```

```
2048x2048: Ref: 115.0765 s, This: 0.4807 s, speedup: 239.38
```

*Wow!*

*But can we be even better?*

Identify available parallelism



Parallelize loops with OpenACC



Optimize data locality



**Optimize loop performance**

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

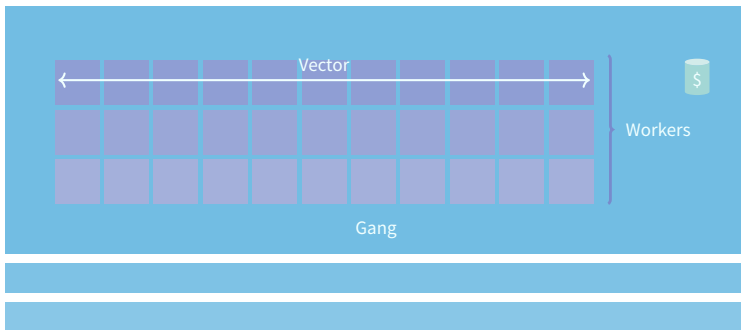
```
110 #pragma acc parallel loop reduction(max:error)
111 for (int ix = ix_start; ix < ix_end; ix++)
112 {
113     // Inner loop
114     for (int iy = iy_start; iy < iy_end; iy++)
115     {
116         Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] - ( A[iy*nx+ix+1] +
117             ↪ A[iy*nx+ix-1] + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix] ));
118         error = fmaxr( error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
119     }
120 }
```

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

- Outer loop: Parallelism with gang and vector
- Inner loop: Sequentially per thread (#pragma acc loop seq)
- Inner loop was never parallelized!
- **Rule of thumb:** Expose as much parallelism as possible

# OpenACC Parallelism

## 3 Levels of Parallelism



### Vector

Vector threads work in lockstep (SIMD/SIMT parallelism)

### Worker

Has 1 or more vector; workers share common resource (*cache*)

### Gang

Has 1 or more workers; multiple gangs work independently from each other

# CUDA Parallelism

## CUDA Execution Model

### Software



Thread

### Hardware



Scalar  
Processor

- **Threads** executed by scalar processors (*CUDA cores*)



### Software



Thread



Thread Block

### Hardware



Scalar  
Processor



Multiprocessor

- **Threads** executed by scalar processors (*CUDA cores*)
- Thread **blocks**: Executed on multiprocessors (*SM*)
- Do not migrate
- Several concurrent thread blocks can reside on multiprocessor  
Limit: Multiprocessor resources (register file; shared memory)

# CUDA Parallelism

## CUDA Execution Model

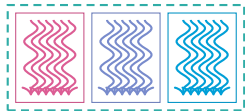
### Software



Thread



Thread Block



Grid

### Hardware



Scalar  
Processor



Multiprocessor



Device

- **Threads** executed by scalar processors (*CUDA cores*)
- Thread **blocks**: Executed on multiprocessors (*SM*)
- Do not migrate
- Several concurrent thread blocks can reside on multiprocessor  
Limit: Multiprocessor resources (register file; shared memory)
- Kernel launched as **grid** of thread blocks
- Blocks, grids: Multiple dimensions

# From OpenACC to CUDA

*map*( $||_{acc}, ||_{<<>>>}$ )

- In general: Compiler free to do what it thinks is best
- Usually
  - gang** Mapped to blocks (*coarse grain*)
  - worker** Mapped to threads (*fine grain*)
  - vector** Mapped to threads (*fine SIMD/SIMT*)
  - seq** No parallelism; sequential
- Exact mapping compiler dependent
- Performance tips
  - Use vector size divisible by 32
  - Block size: `num_workers × vector_length`

# Declaration of Parallelism

*Specify configuration of threads*

- Three **clauses** of parallel region (parallel, kernels) for changing distribution/configuration of group of threads
- Presence of keyword: Distribute using this level
- Optional size: Control size of parallel entity

🔧 OpenACC: gang worker vector

```
#pragma acc parallel loop gang vector
```

Also: worker

Size: num\_gangs(n), num\_workers(n), vector\_length(n)

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

- Compiler reports configuration of parallel entities
  - Gang mapped to `blockIdx.x`
  - Vector mapped to `threadIdx.x`
  - Worker not used
- Here: 128 threads per block; as many blocks as needed  
*128 seems to be default for Tesla/NVIDIA*

# More Parallelism

*Unsequentialize inner loop*

## TASK 6

- Add vector clause to inner loop
- Study result with profiler

### Task 6: More Parallelism

- Change to Task6/ directory
- Work on TODO
- Compile: `make`
- Submit to the batch system: `make run`
- Generate profile with `make profile_tofile`

# More Parallelism

## Compiler Output

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.c
poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    104, Generating create(Anew[:ny*nx])
        Generating copyin(rhs[:ny*nx])
        Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
        Generating Tesla code
    110, Generating reduction(max:error)
    111, #pragma acc loop gang /* blockIdx.x */
    114, #pragma acc loop vector(128) /* threadIdx.x */
    ...
```

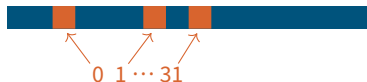
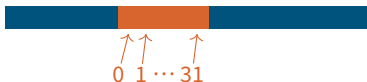
```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 111.7712 s, This:   0.9257 s, speedup:   120.74
```



```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on jura>>
Jacobi relaxation 2048 x 2048 mesh
Calculate reference U execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 111.7712 s, This: 0.9257 s, speedup: 120.74
```

*Actually slower!  
Why?*

- Coalesced access *good*
  - Threads of warp (group of 32 contiguous threads) access adjacent words
  - Few transactions, high utilization
- Uncoalesced access *bad*
  - Threads of warp access scattered words
  - Many transactions, low utilization
- Best **performance**: `threadIdx.x` should access contiguously



# Jacobi's Access Pattern

*Coalesced or uncoalesced, that is the question*

```
#pragma acc parallel loop reduction(max:error)
for (int ix = ix_start; ix < ix_end; ix++) {
    #pragma acc loop vector
    for (int iy = iy_start; iy < iy_end; iy++) {
        Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
            ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
              + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
        //...
```

- Fast-running index: ix
- Slow-running index: iy
- But vector loop over iy!
- Consecutive threads access far away memory location!

- Interchange loop order for Jacobi loops
- Also: Compare to loop-fixed CPU reference version

### Task 7: Loop Ordering

- Change to Task7/ directory
- Work on TODO
- Compile: `make`
- Submit to the batch system: `make run`

# Fixing Access Pattern

*Compiler output (unchanged)*

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.c
poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    104, Generating create(Anew[:ny*nx])
        Generating copyin(rhs[:ny*nx])
        Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
        Generating Tesla code
    110, Generating reduction(max:error)
    111, #pragma acc loop gang /* blockIdx.x */
    114, #pragma acc loop vector(128) /* threadIdx.x */
    ...
```

# Fixing Access Pattern

## Run Result

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 113.0214 s, This:   0.3284 s, speedup:   344.15
```

# Fixing Access Pattern

## Run Result

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
```

Ja sh

Ca *Again with proper CPU version!*

```
100, 0.249760
```

```
200, 0...
```

```
Calculate current execution.
```

```
0, 0.249999
```

```
100, 0.249760
```

```
200, 0...
```

```
2048x2048: Ref: 113.0214 s, This: 0.3284 s, speedup: 344.15
```

# Fixing Access Pattern

## Run Result II

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref:    6.8080 s, This:    0.2609 s, speedup:    26.10
```



# Fixing Access Pattern

## Run Result II

```
$ make run
bsub -I -R "rusage[ngpus_shared=20]" ./poisson2d
Job <4490> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation on 2048 x 2048 mesh
Calculate reference CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref:    6.8080 s, This:    0.2609 s, speedup:    26.10
```

*26 × is great!*

- Host memory allocated with `malloc()` is pageable
  - Memory pages of memory can be moved by kernel, e.g. swapped to disk
  - Additional indirection

- Host memory allocated with `malloc()` is pageable
  - Memory pages of memory can be moved by kernel, e.g. swapped to disk
  - Additional indirection
- NVIDIA GPUs can allocate **page-locked memory** (*pinned* memory)
  - + Faster (safety guards are skipped)
  - + Interleaving of execution and copy (asynchronous)
  - + Directly map into GPU memory\*
  - Scarce resource; OS performance could degrade

- Host memory allocated with `malloc()` is pageable
  - Memory pages of memory can be moved by kernel, e.g. swapped to disk
  - Additional indirection
- NVIDIA GPUs can allocate **page-locked memory** (*pinned* memory)
  - + Faster (safety guards are skipped)
  - + Interleaving of execution and copy (asynchronous)
  - + Directly map into GPU memory\*
  - Scarce resource; OS performance could degrade
- OpenACC: Very easy to use pinned memory
  - ta=tesla:pinned

- Compare performance with and without pinned memory
- Also test unified memory again

## Task 7': Pinned Memory

- Like in Task 7, but change compilation to include pinned or managed
- Submit to the batch system: `make run`

**Identify available parallelism**



**Parallelize loops with OpenACC**



**Optimize data locality**



**Optimize loop performance**

# Interoperability

- OpenACC can operate together with
  - Applications
  - Libraries
  - CUDA



```
host_data use_device
```

## host\_data use\_device

- Background
    - GPU and CPU are different devices, have different memory
    - Distinct address spaces
  - OpenACC hides handling of addresses from user
    - For every chunk of accelerated data, **two** addresses exist
    - One for CPU data, one for GPU data
    - OpenACC uses appropriate address in accelerated kernel
  - **But:** Automatic handling not working when out of OpenACC (OpenACC will default to host address)
- **host\_data use\_device** uses the address of the GPU device data for scope

# The host\_data Construct

*That's all you need*

- Usage:

```
double* foo = new double[N];           // foo on Host
#pragma acc data copyin(foo[0:N])      // foo on Device
{
    ...
    #pragma acc host_data use_device(foo)
    some_lfunc(foo);                    // Device: OK!
    ...
}
```

- Directive can be used for structured block as well

# The Inverse: deviceptr

*When CUDA is involved*

- For the inverse case:
  - Data has been copied by CUDA or a CUDA-using library
  - Pointer to data residing on devices is returned
  - Use this data in OpenACC context
- deviceptr clause declares data to be on device

# The Inverse: deviceptr

*When CUDA is involved*

- For the inverse case:
  - Data has been copied by CUDA or a CUDA-using library
  - Pointer to data residing on devices is returned
  - Use this data in OpenACC context
- deviceptr clause declares data to be on device
- Usage:

```
float * n;  
int n = 4223;  
cudaMalloc((void**)&x, (size_t)n*sizeof(float));  
// ...  
#pragma acc kernels deviceptr(x)  
for (int i = 0; i < n; i++) {  
    x[i] = i;  
}
```

# Interoperability

## Tasks

# Task 1

## Introduction to BLAS

- Use case: Anything linear algebra
- **BLAS**: Basic Linear Algebra Subprograms
  - Vector-vector, vector-matrix, matrix-matrix operations
  - Specification of routines
  - Examples: SAXPY, DGEMV, ZGEMM→ <http://www.netlib.org/blas/>
- **cuBLAS**: NVIDIA's linear algebra routines with BLAS interface, readily accelerated  
→ <http://docs.nvidia.com/cuda/cublas/>
- **Task 1**: Use cuBLAS for vector addition, everything else with OpenACC

## Task 8-1

### *cuBLAS OpenACC Interaction*

- cuBLAS routine used:

```
cublasDaxpy(cublasHandle_t handle, int n,  
            const double          *alpha,  
            const double          *x, int incx,  
            double                *y, int incy)
```

- handle capsules GPU auxiliary data, needs to be created and destroyed with `cublasCreate` and `cublasDestroy`
- `x` and `y` point to addresses on **device**!
- cuBLAS library needs to be linked with `-lcublas`



## Task 8-1

### *Vector Addition with cuBLAS*

#### TASK 8-1

- Use cuBLAS for vector addition

#### Task 8-1: OpenACC +cuBLAS

- Change to Task8-1/ directory
- Work on TODOs in `vecAddRed.c`
  - Use `host_data use_device` to provide correct pointer
  - Check [cuBLAS documentation](#) for details on `cublasDaxpy()`
- Compile: `make`
- Submit to the batch system: `make brun`

## Task 8-2

### CUDA Need-to-Know

- Use case:
    - Working on legacy code
    - Need the *raw* power (/flexibility) of CUDA
  - CUDA need-to-knows:
    - Thread → Block → Grid  
*Total* number of threads should map to your problem; threads are always given per block
    - A kernel is called from every thread on GPU device  
Number of kernel threads: *triple chevron syntax*  
`kernel<<<nBlocks, nThreads>>>(arg1, arg2, ...)`
    - Kernel: Function with `__global__` prefix  
Aware of its index by global variables, e.g. `threadIdx.x`
- <http://docs.nvidia.com/cuda/>

## Task 8-2

### *Vector Addition with CUDA Kernel*

#### TASK 8-2

- CUDA kernel for vector addition, rest OpenACC
- Marrying CUDA C and OpenACC:
  - All direct CUDA interaction wrapped in wrapper file `cudaWrapper.cu`, compiled with `nvcc` to object file (`-c`)
  - `vecAddRed.c` calls external function from `cudaWrapper.cu` (**extern**)

#### Task 8-2: OpenACC +CUDA

- Change to `Task8-2/` directory
- Work on TODOs in `vecAddRed.c` and `cublasWrapper.cu`
  - Use `host_data use_device` to provide correct pointer
  - Implement computation in kernel, implement call of kernel
- Compile: `make`
- Submit to the batch system: `make brun`

## Task 8-3

### *Vector Addition with Thrust*

- **Thrust**
  - Template library for CUDA C/C++ (similar to STL)
  - Offers many pre-made algorithms for popular computing tasks
  - Usually works with C++ iterators, but understands C arrays as well
  - <http://thrust.github.io/>
- Use Thrust for reduction, everything else of vector addition with OpenACC

#### Task 8-3: OpenACC +CUDA

- Change to Task8-3/ directory
- Work on TODOs in `vecAddRed.c` and `thrustWrapper.cu`
  - Use `host_data` `use_device` to provide correct pointer
  - Implement call to `thrust::reduce` using `C_ptr`
- Compile: `make`
- Submit to the batch system: `make brun`

## Task 8-4

### *Stating the Problem*

- We want to solve the Poisson equation

$$\Delta\Phi(x,y) = -\rho(x,y)$$

with periodic boundary conditions in  $x$  and  $y$

- Needed, e.g., for finding electrostatic potential  $\Phi$  for a given charge distribution  $\rho$
- Model problem

$$\begin{aligned}\rho(x,y) &= \cos(4\pi x) \sin(2\pi y) \\ (x,y) &\in [0,1)^2\end{aligned}$$

- Analytically known:  $\Phi(x,y) = \Phi_0 \cos(4\pi x) \sin(2\pi y)$
- Let's solve the Poisson equation with a Fourier Transform!

## Task 8-4

### Introduction to Fourier Transforms

- Discrete Fourier Transform and Re-Transform:

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j e^{-\frac{2\pi i k}{N} j} \quad \Leftrightarrow \quad f_j = \sum_{k=0}^{N-1} \hat{f}_k e^{\frac{2\pi i j}{N} k}$$

- Time for all  $\hat{f}_k$ :  $\mathcal{O}(N^2)$
- Fast Fourier Transform: Recursively splitting  $\rightarrow \mathcal{O}(N \log(N))$
- Find derivatives in Fourier space:

$$f'_j = \sum_{k=0}^{N-1} i k \hat{f}_k e^{\frac{2\pi i j}{N} k}$$

*It's just multiplying by  $ik$ !*

## Task 8-4

### Plan for FFT Poisson Solution

Start with charge density  $\rho$

- 1 Fourier-transform  $\rho$   
 $\hat{\rho} \leftarrow \mathcal{F}(\rho)$
- 2 Integrate  $\rho$  in Fourier space twice  
 $\hat{\phi} \leftarrow -\hat{\rho} / (k_x^2 + k_y^2)$
- 3 Inverse Fourier-transform  $\hat{\phi}$   
 $\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$

## Task 8-4

### Plan for FFT Poisson Solution

Start with charge density  $\rho$

- 1 Fourier-transform  $\rho$

$$\hat{\rho} \leftarrow \mathcal{F}(\rho)$$

cuFFT

- 2 Integrate  $\rho$  in Fourier space twice

$$\hat{\phi} \leftarrow -\hat{\rho} / (k_x^2 + k_y^2)$$

OpenACC

- 3 Inverse Fourier-transform  $\hat{\phi}$

$$\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$$

cuFFT



## Task 8-4

### cuFFT

- cuFFT: NVIDIA's (Fast) Fourier Transform library
    - 1D, 2D, 3D transforms; complex and real data types
    - Asynchronous execution
    - Modeled after FFTW library (API)
    - Part of CUDA Toolkit
- <https://developer.nvidia.com/cufft>

```
cufftDoubleComplex *src, *tgt;           // Device data!
cufftHandle plan;
// Setup 2d complex-complex trafo w/ dimensions (Nx, Ny)
cufftCreatePlan(plan, Nx, Ny, CUFFT_Z2Z);
cufftExecZ2Z(plan, src, tgt, CUFFT_FORWARD); // FFT
cufftExecZ2Z(plan, tgt, tgt, CUFFT_INVERSE); // iFFT
// Inplace trafo ^-----^
cufftDestroy(plan);                      // Clean-up
```

## Task 8-4

### Synchronizing cuFFT

- CUDA Streams enable interleaving of computational tasks
- cuFFT uses streams for asynchronous execution
- cuFFT runs in default CUDA stream;  
OpenACC not → trouble

⇒ Force cuFFT on OpenACC stream

```
#include <openacc.h>
// Obtain the OpenACC default stream id
cudaStream_t accStream =
    (cudaStream_t) acc_get_cuda_stream(acc_async_sync) ;
// Execute all cufft calls on this stream
cufftSetStream(accStream);
```

## Task 8-4

*OpenACC and cuFFT*

### TASK 8-4

- Use case: Fourier transforms
- Use cuFFT and OpenACC to solve Poisson's Equation

#### Task 8-4: OpenACC +cuFFT

- Change to Task8-4/ directory
- Work on TODOs in `poisson.c`
  - `solveRSpace` Force cuFFT on correct stream; implement data handling with `host_data use_device`
  - `solveKSpace` Implement data handling and parallelism
- Compile: `make`
- Submit to the batch system: `make brun`

# Conclusions

- OpenACC directives and clauses  
`#pragma acc parallel loop copyin(A[0:N])  
reduction(max:err) vector`
- Start easy, optimize from there
- PGI / NVIDIA Visual Profiler help to find bottlenecks
- OpenACC is interoperable to other GPU programming models
- Don't forget the CPU version!

- OpenACC directives and clauses  
`#pragma acc parallel loop copyin(A[0:N])  
reduction(max:err) vector`
- Start easy, optimize from there
- PGI / NVIDIA Visual Profiler help to find bottlenecks
- OpenACC is interoperable to other GPU programming models
- Don't forget the CPU version!

**Thank you  
for your attention!**  
[a.herten@fz-juelich.de](mailto:a.herten@fz-juelich.de)

# Appendix

- List of Tasks
- Glossary
- References

# List of Tasks

Task 0\*: Setup

Task 0: Getting Started

Task 1: Analyze Application

Task 2: A First Parallel Loop

Task 3: More Parallel Loops

Task 4: Data Copies

Task 5: Data Region

Task 6: More Parallelism

Task 7: Loop Ordering

Task 7': Pinned Memory

Task 8-1: OpenACC +cuBLAS

Task 8-2: OpenACC +CUDA

Task 8-3: OpenACC +CUDA

Task 8-4: OpenACC +cuFFT



**API** A programmatic interface to software by well-defined functions. Short for application programming interface. [79](#)

**CUDA** Computing platform for [GPUs](#) from NVIDIA. Provides, among others, CUDA C/C++. [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [33](#), [46](#), [79](#), [93](#), [94](#), [95](#), [96](#), [131](#), [152](#), [156](#), [157](#), [162](#), [163](#), [164](#), [169](#), [170](#), [176](#), [177](#)

**GCC** The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. [45](#), [49](#)

**NVIDIA** US technology company creating GPUs. 39, 79, 92, 97, 98, 105, 106, 146, 147, 148, 159, 169, 173, 174, 177

**OpenACC** Directive-based programming, primarily for many-core machines. 2, 33, 38, 39, 40, 41, 42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 62, 69, 70, 72, 74, 76, 79, 82, 87, 97, 98, 99, 100, 108, 117, 124, 127, 131, 132, 146, 147, 148, 150, 152, 153, 154, 156, 157, 159, 160, 161, 163, 164, 167, 168, 170, 171, 173, 174, 176

**OpenCL** The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 33

- OpenMP** Directive-based programming, primarily for multi-threaded machines. 2, 33, 40, 41, 42, 50, 84, 85, 86
- Pascal GPU** architecture from **NVIDIA** (announced 2016). 93, 94, 95, 96
- Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 33
- CPU** Central Processing Unit. 4, 5, 6, 7, 8, 9, 10, 11, 12, 45, 50, 79, 93, 94, 95, 96, 140, 143, 153, 154, 173, 174, 177

**GPU** Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 23, 24, 25, 26, 33, 45, 48, 50, 70, 76, 79, 92, 93, 94, 95, 96, 97, 98, 105, 115, 116, 118, 146, 147, 148, 153, 154, 160, 162, 173, 174, 177

- [3] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: <http://doi.acm.org/10.1145/1465482.1465560>.
- [4] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: <http://doi.acm.org/10.1145/42411.42415>.

- [1] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (pages 4, 5).
- [2] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (pages 4, 5).