

# PROGRAMMING GPU-ACCELERATED OPENPOWER SYSTEMS WITH OPENACC

## GPU TECHNOLOGY CONFERENCE 2018

26 March 2018 | Andreas Herten | Forschungszentrum Jülich *Handout Version*

# Overview, Outline

## What you will learn today

- What's special about GPU-equipped POWER systems
- Parallelization strategies with OpenACC
- OpenACC on CPU, GPU, GPUs
- *All in 120 minutes*

## What you will not learn today

- Analyze program in-detail
- Strategies for complex programs
- How to leave the matrix

Introduction

POWER

Login **E**

OpenACC Introduction

OpenACC on CPU **E**

OpenACC: GPU Optimizations

OpenACC with GPUs **E**

MPI 101

OpenACC, GPUs, and MPI **E**

Hands-on

Lecture

Extra

# Overview, Outline

## What you will learn today

- What's special about GPU-equipped POWER systems
- Parallelization strategies with OpenACC
- OpenACC on CPU, GPU, GPUs
- *All in 120 minutes*

## What you will not learn today

- Analyze program in-detail
- Strategies for complex programs
- How to leave the matrix

## Introduction

### OpenPOWER

Minsky, POWER8

Newell, POWER9

Using JURON

### OpenACC Introduction

About OpenACC

Modus Operandi

OpenACC's Models

Parallelization Workflow

### First Steps in OpenACC

Example Program

Identify Parallelism

Parallelize Loops

parallel

loops

kernels

## OpenACC on the GPU

Compiling on GPU

Data Locality

copy

data

enter data

## OpenACC on Multiple GPUs

MPI 101

Jacobi MPI Strategy

Asynchronous

## Conclusions, Summary

Appendix

List of Tasks

# Jülich

## Jülich Supercomputing Centre

- Forschungszentrum Jülich: One of largest research centers in Europe
- Jülich Supercomputing Centre: Host of and research in supercomputers  
JUQUEEN BlueGene/Q system, †Mar 2018, then: JUWELS  
JURECA Intel x86 system; some GPUs, many KNLs  
etc DEEP, QPACE, JULIA, JURON
- Me: Physicist, now at *POWER Acceleration and Design Centre* and *NVIDIA Application Lab*

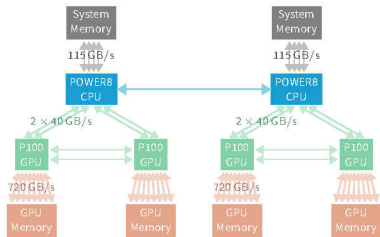


- Platform for collaboration around POWER processor architecture
- Started by IBM, NVIDIA, many more (now > 250 members)
- Objectives
  - Licensing of processor architecture to partners
  - Collaborate on system extension
  - Open-Source Software
- Example technology: NVLink, fast GPU-CPU interconnect

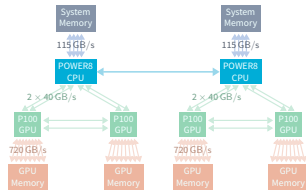
→ <https://openpowerfoundation.org/>

# Minsky System

- IBM's S822LC server, codename *Minsky*
- 2 IBM POWER8NVL CPUs, 4 NVIDIA Tesla P100 GPUs



# System Core Numbers



## POWER8 CPU

- 2 sockets, each 10 cores, each  $8 \times$  SMT
- 2.5 GHz to 5 GHz; 8 FLOP/Cycle/Core
- 256 GB memory (115 GB/s)
- L4 \$ per socket:  $4 \times 16$  MB (Buffer Chip)
- L3, L2, L1 \$ per core: 8 MB, 512 kB, 64 kB

0.5 TFLOP/s

## P100 GPU

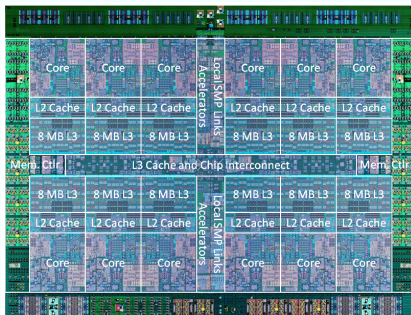
- 56 Streaming Multiprocessors (SMs)
- 64 FLOP/Cycle/SM
- 16 GB (720 GB/s)
- L2 \$: 4 MB
- Shared Memory: 64 kB

5 TFLOP/s

NVLink (40 GB/s)

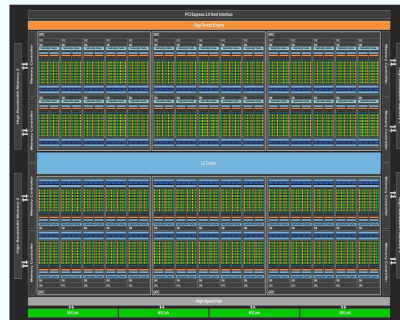
# System Core Numbers

## POWER8 CPU



0.5 TFLOP/s

## P100 GPU



5 TFLOP/s

NVLink (40 GB/s)

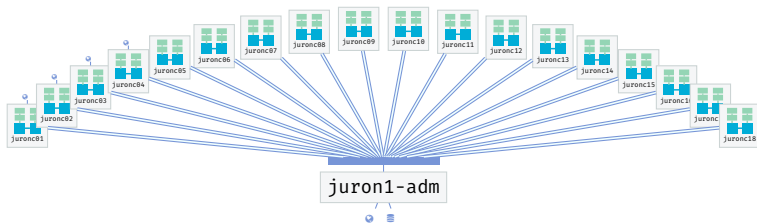


# JURON



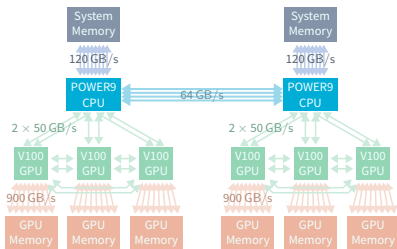
## JURON (Juelich + Neuron)

- 18 Minsky nodes ( $\approx 350$  TFLOP/s)
- For Human Brain Project (HBP), but not only
- Prototype system, together with JULIA (KNL-based)
- **Access via Jupyter Hub or SSH**



# Newell

- Successor of Minsky (AC922 instead of S822LC)
  - POWER9 instead of POWER8, 3 (2) Voltas instead of 2 Pascals, NVLink 2 instead of NVLink 1
- Faster memory bandwidths, more FLOP/s, smarter NVLink



→ Appendix 1, 2

## Tesla V100

- 80 SMs
- FP32, FP64 cores per SM same as Pascal ⇒ 7.5 TFLOP (FP64)/ sec
- 8 Tensor Cores per SM ⇒ 120 TFLOP (FP16)/ sec
- NVLink 2: Cache coherence, ...; CPU Address Translation Service

# Summit

- New supercomputer at [Oak Ridge National Lab](#)
- 4600 Newell-like nodes
- $> 200$  PFLOP/s performance
- Maybe the world's fastest supercomputer!
- Also: [Sierra](#) at Lawrence Livermore National Laboratory



### Task 1: JURON

- Website of Lab: <http://bit.ly/gtc18-openacc>
- Log in to JURON via <http://jupyter-jsc.fz-juelich.de>
  - Access via Jupyter Lab (*no Notebooks, but Terminal*)
  - Login from slip of paper (»Workshop password«)
  - Click through to launch Jupyter Lab instance on JURON
  - Start Terminal, browse to source files, view slides, ...
- Directory of tasks `cd $HOME/Tasks/Tasks/`
- Solutions are always given! You decide when to look
- Edit files with Jupyter's source code editor (just open `.c` file)

? How many cores are on a compute node? How many CUDA cores? See `README.md`

# Using JURON

## A gentle start

### TASK 1

#### Task 1: JURON

- Website of Lab: <http://www.jurion.de>
- Log in to JURON via
  - Access via Jupyter
  - Login from slip c
  - Click through to
  - Start Terminal, b
- Directory of tasks [cd](#)
- Solutions are alway
- Edit files with Jupyter

? How many cores are



[bit.ly/gtc18-openacc](http://bit.ly/gtc18-openacc)

.de

e)

cores? See README.md

## Task 1

- W
- L

- D
- S
- E

? H

JupyterHub

Sicher | <https://jupyter-jsc.fz-juelich.de/hub/login>

jupyter

**Sign In with JSC Webservice Account**

[Register for a JSC Webservice Account](#)

## Welcome to Jupyter@JSC

### News

**26.02.2018**

- Workshop users are supported now.  
For this you have to write a email ([jupyter.jsc@fz-juelich.de](mailto:jupyter.jsc@fz-juelich.de)) and ask for access for training accounts.

**22.02.2018**

- JupyterLab is supported on JURON.

**15.02.2018**

- JURON was added to the list of supported HPC-Systems.

### Requirements

- You need a [JURECA](#) or a [JURON](#) account.
- You need a [JSC Webservice Account](#)

### Support

- If you find any bugs or if you want to give us feedback: [jupyter.jsc@fz-juelich.de](mailto:jupyter.jsc@fz-juelich.de)

# Using JURON

A gentle sta

## TASK 1

### Task 1

The screenshot displays a JupyterLab environment with the following components:

- File Explorer (Left):** Shows a directory structure under 'Tasks > Task2'. The files listed are 'common.h', 'Makefile', 'poisson2d\_reference.c', 'poisson2d.c' (selected), and 'README.md'. All files were modified '2 minutes ago'.
- Code Editor (Center):** Displays the source code for 'poisson2d.c'. The code is a C program that takes command-line arguments for the number of iterations and a tolerance value, then performs a loop.
- Terminal (Bottom Left):** Shows a terminal window for 'train007@juron'. It displays a maintenance notice: 'JURON will be unavailable on 2018-04-10 07:00 - 10:00 CEST' and a date change from '2018-03-19' to '2018-04-10'. The prompt is '[train007@juron1-adm ~]\$'.
- README.md (Top Right):** Contains instructions for using the tasks, mentioning '[Solutions](../Solutions)' and '[Tasks](../Tasks)'.
- Slides (Bottom Right):** Shows a presentation slide titled 'PROGRAMMING GPU-ACCELERATED OPENPOWER SYSTEMS WITH OPENACC' from the 'GPU TECHNOLOGY CONFERENCE 2018'. The slide features an image of a rocket launch.

# Using JURON

## A gentle start

### TASK 1

#### Task 1: JURON

- Website of Lab: <http://bit.ly/gtc18-openacc>
- Log in to JURON via <http://jupyter-jsc.fz-juelich.de>
  - Access via Jupyter Lab (*no Notebooks, but Terminal*)
  - Login from slip of paper (»Workshop password«)
  - Click through to launch Jupyter Lab instance on JURON
  - Start Terminal, browse to source files, view slides, ...
- Directory of tasks `cd $HOME/Tasks/Tasks/`
- Solutions are always given! You decide when to look
- Edit files with Jupyter's source code editor (just open `.c` file)

? How many cores are on a compute node? How many CUDA cores? See `README.md`



# Using JURON

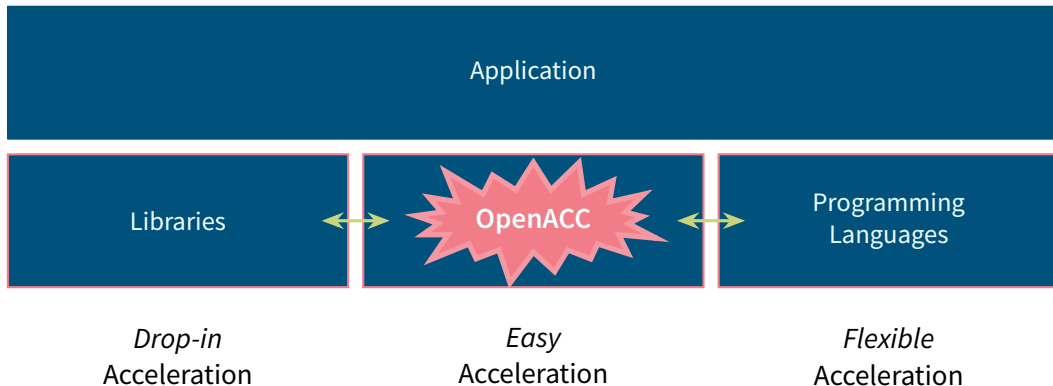
So many cores!

```
$ make run
bsub -Is -U gtc lscpu
[...]
CPU(s):                160
[...]
module load cuda cuda-samples && \
bsub -Is -R "rusage[ngpus_shared=1]" -U gtc deviceQuery
[...]
Device 0: "Tesla P100-SXM2-16GB"
  CUDA Driver Version / Runtime Version      9.1 / 9.1
  CUDA Capability Major/Minor version number: 6.0
  Total amount of global memory:              16276 MBytes (17066885120 bytes)
  (56) Multiprocessors, ( 64) CUDA Cores/MP: 3584 CUDA Cores
  [...]
```

→ Total number of (totally different) cores:  $160 + (4 \times 3584) = 14\,496$

# OpenACC Introduction

# Primer on GPU Computing





# About OpenACC


## History

2011 OpenACC 1.0 specification is released 

*NVIDIA, Cray, PGI, CAPS*

2013 OpenACC 2.0: More functionality, portability 

2015 OpenACC 2.5: Enhancements, clarifications 

2017 OpenACC 2.6: Deep copy, ... 

→ <https://www.openacc.org/> (see also: *Best practice guide*  )

## Support

- Compiler: PGI, GCC, Cray, *Sunway*
- Languages: C/C++, Fortran

# Open{MP↔ACC}

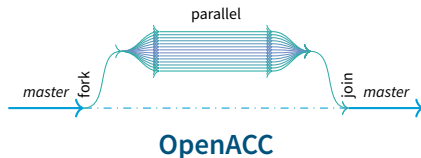
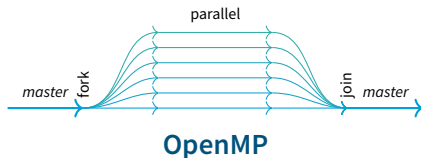
## Everything's connected

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

*But OpenMP >4.0 also has offloading feature*

- OpenACC more descriptive, OpenMP more prescriptive
- Basic principle same: Fork/join model

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



# Modus Operandi

## Three-step program

- 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 many-core machines, especially accelerators
- 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; always 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**

*\*: Eventually you want to tune for device; but that's possible*



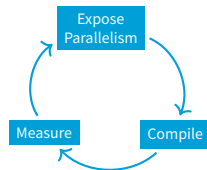
### 3 Success

Iteration is key

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

#### ⇒ Productivity

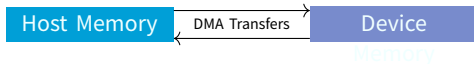
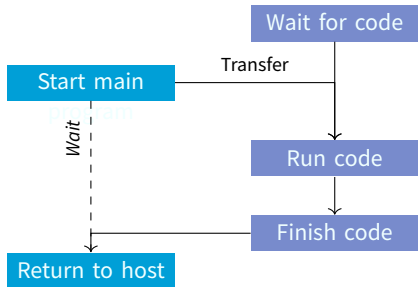
- Because of *generallness*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, ...)



# OpenACC Accelerator Model

For computation and memory spaces

- Main program executes on **host**
- Device code is transferred to **accelerator**
- Execution on accelerator is started
- Host waits until return (except: async)
- Two separate memory spaces; data transfers back and forth
  - Transfers hidden from programmer
  - Memories not coherent!
  - Compiler helps; GPU runtime helps



# OpenACC Programming Model

## A binary perspective

- OpenACC interpretation needs to be activated as compile flag

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

**GCC** `gcc -fopenacc`

→ Ignored by incapable compiler!

- Additional flags possible to improve/modify compilation

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

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

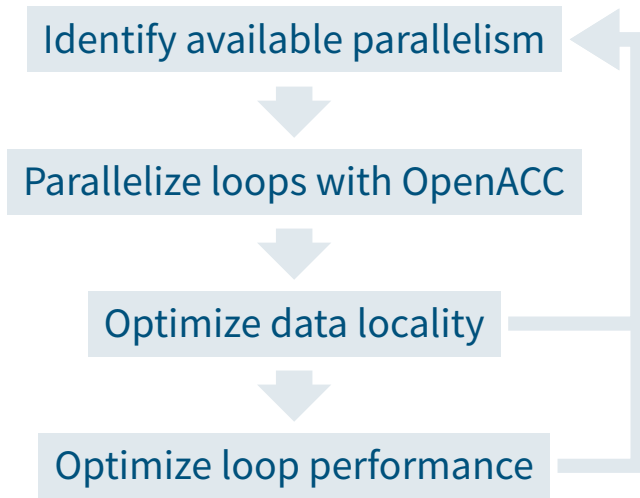
`-ta=tesla:managed` Use unified memory

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

# A Glimpse of OpenACC

```
#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];
    }
}
```

# Parallelization Workflow

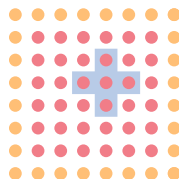
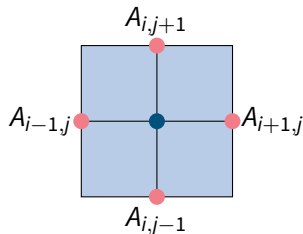


# First Steps in OpenACC

# Jacobi Solver

## Algorithmic description

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



- Data Point
- Boundary Point
- Stencil

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

# Jacobi Solver

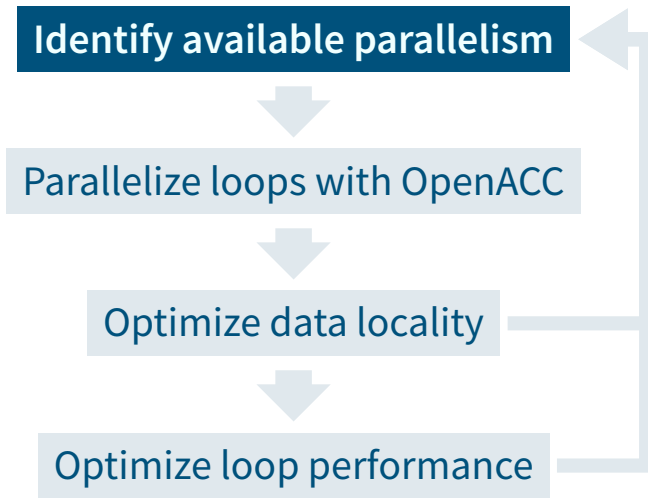
## Source code

```
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



# Parallelization Workflow



# Profiling

## Profile

*[...] premature optimization is the root of all evil.*

***Yet we should not pass up our [optimization] opportunities [...]***

*– Donald Knuth [10]*

- Investigate hot spots of your program!

→ Profile!

- Many tools, many levels: perf, PAPI, Score-P, Intel Advisor, NVIDIA Visual Profiler, ...
- Here: Examples from PGI

# 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

```
$ pgprof --cpu-profiling on [...] ./poisson2d
===== 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 Independency Analysis

Independence is key

```
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++;  
}
```

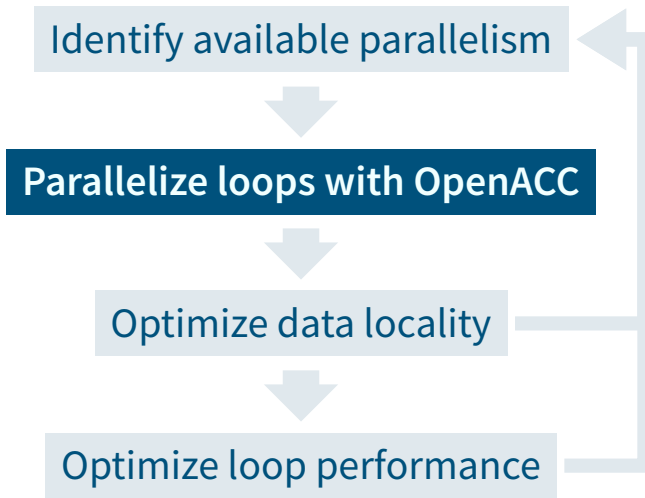
Data dependency  
between iterations

Independent loop  
iterations

Independent loop  
iterations

Independent loop  
iterations

# Parallelization Workflow



# Parallel Loops: Parallel

Maybe the second most important directive

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

🔗 OpenACC: parallel

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

# Parallel Loops: Parallel

## Clauses

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}
```

# Parallel Loops: Loops

## Clauses

`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;  
}
```

Kernel 1

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

Kernel 2

## Add parallelism

- Add OpenACC parallelism to main loop in Jacobi solver source code (CPU parallelism)
- Congratulations, you are a parallel developer!

### Task 2: A First Parallel Loop

- Change to Task2/ directory
  - Compile: `make`; see `README.md`
  - Submit run to the batch system: `make run`  
*Adapt the `bsub` call and run with other number of iterations, matrix sizes*
  - Change number of CPU threads via `$ACC_NUM_CORES` or `$OMP_NUM_THREADS`
- ? What's your speed-up? What's the best configuration for cores?

**E** Compare it to OpenMP

# Parallel Jacobi

## Source Code

```
110  #pragma acc parallel loop reduction(max:error)
111  for (int ix = ix_start; ix < ix_end; ix++)
112  {
113      for (int iy = iy_start; iy < iy_end; iy++)
114      {
115          Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] - ( A[iy*nx+ix+1] +
116              ↪ A[iy*nx+ix-1]
117                  + A[(iy-1)*nx+ix] +
118                  ↪ A[(iy+1)*nx+ix] ));
119          error = fmaxr( error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
120      }
121  }
```

# Parallel Jacobi

## Compilation result

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=multicore poisson2d.c poisson2d_reference.o
-o poisson2d
poisson2d.c:
main:
  110, Generating Multicore code
    111, #pragma acc loop gang
  110, Generating reduction(max:error)
  113, Accelerator restriction: size of the GPU copy of A,rhs,Anew is unknown
      Complex loop carried dependence of Anew-> prevents parallelization
      Loop carried dependence of Anew-> prevents parallelization
      Loop carried backward dependence of Anew-> prevents vectorization
```

# Parallel Jacobi

## Run result

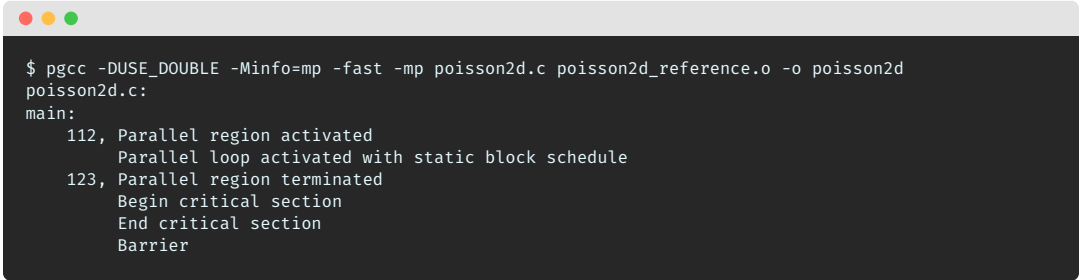
```
$ make run
bsub -I -R "rusage[ngpus_shared=1]" -U gtc ./poisson2d
Job <4444> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref:  56.6275 s, This:  19.9486 s, speedup:    2.84
```



- OpenMP pragma is quite similar

```
#pragma acc parallel loop reduction(max:error)
#pragma omp parallel for reduction(max:error)
for (int ix = ix_start; ix < ix_end; ix++) { ... }
```

- PGI's compiler output is a bit different (but states the same)

A terminal window with a grey title bar and three colored window control buttons (red, yellow, green) on the left. The terminal has a dark background and displays the output of a PGI compiler command. The output shows the activation and termination of a parallel region, followed by critical sections and a barrier.

```
$ pgcc -DUSE_DOUBLE -Minfo=mp -fast -mp poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
  112, Parallel region activated
      Parallel loop activated with static block schedule
  123, Parallel region terminated
      Begin critical section
      End critical section
      Barrier
```

- Run time should be very similar!

# 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] ...]
```

# Kernels Example

```
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

# kernels vs. parallel

- 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
  - More explicit
  - 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

# OpenACC on the GPU

# Changes for GPU-OpenACC

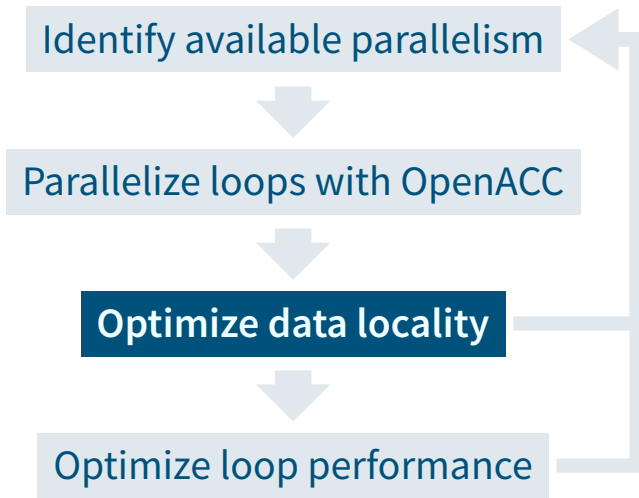
Immensely complicated changes

- Necessary for previous code to run on GPU: `-ta=tesla` instead of `-ta=multicore`

⇒ **That's it!**

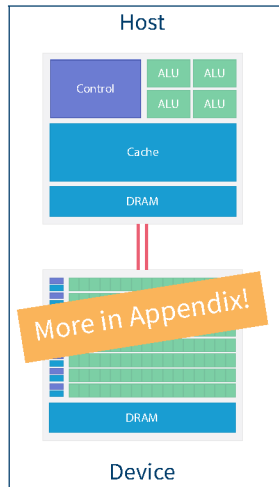
- But we can optimize!

# Parallelization Workflow



# Automatic Data Transfers

- Up to now: We did not care about **data transfers**
- Compiler and runtime care
- CPU data can be copied automatically to GPU via Managed Memory
- Magic keyword: `-ta=tesla:managed`
- Be more explicit for full portability and full performance





# Copy Clause

- Explicitly inform OpenACC compiler about data intentions
- Use data which is already on GPU; only copy parts of it; ...

## 🔗 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 Regions

To manually specify data locations: `data` construct

- 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] ...]
```

# Data Regions

## Clauses

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

# Data Region Example

```
#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];
  }
}
```

# Data Regions II

Looser regions: `enter` data directive

- Define data regions, but not for structured block
- Closest to `cudaMemcpy()`
- Still, explicit data transfers

🔗 OpenACC: `enter` data

```
#pragma acc enter data [clause, [, clause] ...]  
#pragma acc exit data [clause, [, clause] ...]
```

# Parallel Jacobi II

## More parallelism, Data locality

### TASK 3

- Add OpenACC parallelism to other loops of `while` (L:123 – L:141)  
*Use either `kernels` or `parallel`*
- Add data regions such that all data resides on device during iterations

### Task 3: More Parallel Loops

- Change to Task3/ directory
- Change source code; see README.md
- Compile: `make`
- Submit parallel run to the batch system: `make run`

? What's your speed-up?

**E** Change order of for loop!

# Parallel Jacobi II

## Source Code

```
105 #pragma acc data copy(A[0:nx*ny]) copyin(rhs[0:nx*ny]) create(Anew[0:nx*ny])
106 while ( error > tol && iter < iter_max )
107 {
108     error = 0.0;
109
110     // Jacobi kernel
111     #pragma acc parallel loop reduction(max:error)
112     for (int ix = ix_start; ix < ix_end; ix++)
113     {
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] + A[iy*nx+ix-1]
117                                                         + 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     }
121
122     // A <-> Anew
123     #pragma acc parallel loop
124     for (int iy = iy_start; iy < iy_end; iy++)
125     // ...
126 }
```

# Parallel Jacobi II

## Compilation result

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

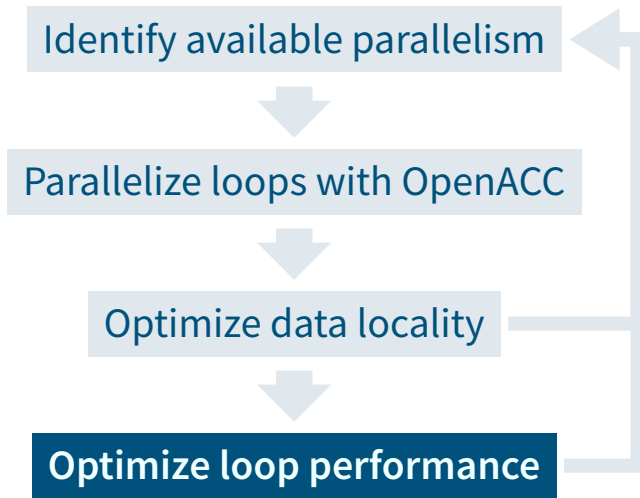


# Parallel Jacobi II

## Run result

```
$ make run
bsub -I -R "rusage[ngpus_shared=1]" ./poisson2d
Job <4444> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc10>>
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:  53.7294 s, This:  0.3775 s, speedup:  142.33
```

# Parallelization Workflow



# Parallel Jacobi II+

E

## Expert Task

Improve memory access pattern: Loop order in main loop

```
#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] =
```

```
        ↪ (rhs[iy*nx + ix] +
```

```
            ( A[iy*nx + ix-1]
```

```
            + A[(iy-1)*nx + ix]
```

```
        //...
```

ix Outer loop index; accesses

memory locations

ix; accesses offset

locations

change order to optimize pattern ✓

More on OpenACC thread  
configuration in Appendix!

# Parallel Jacobi II+

E

## Expert Task

Improve memory access pattern: Loop order in main loop

```
#pragma acc parallel loop reduction(max:error)
```

```
for (int iy = iy_start; iy < iy_end; iy++) {
```

```
    #pragma acc loop vector
```

```
    for (int ix = ix_start; ix < ix_end; ix++) {
```

```
        Anew[iy*nx + ix] =
```

```
        ↪ (rhs[iy*nx + ix] +
```

```
            ( A[iy*nx + ix] +
```

```
              + A[(iy-1)*nx + ix] +
```

```
              //...
```

ix Outer loop index; accesses

memory locations

ix; accesses offset

locations

change order to optimize pattern ✓

More on OpenACC thread  
configuration in Appendix!

# Parallel Jacobi II+

E

## Expert Task

Improve memory access pattern: Loop order in main loop

```
#pragma acc parallel loop reduction(max:error)
```

```
for (int iy = iy_start; iy < iy_end; iy++) {
```

```
    #pragma acc loop vector
```

```
    for (int ix = ix_start; ix < ix_end; ix++) {
```

```
        Anew[iy*nx +
```

```
            ↪ (rhs[iy*nx +
```

```
                ( A[iy*nx +
```

```
                    + A[(iy-1)*nx
```

```
//...
```

ix Outer loop index; accesses

memory locations

ix; accesses offset

locations

change order to optimize pattern ✓

More on OpenACC thread  
configuration in Appendix!

```
$ make run
```

```
[...]
```

```
2048x2048: Ref: 69.0022 s, This: 0.2680 s, speedup: 257.52
```

# Parallel Jacobi II+

E

## Expert Task

Improve memory access pattern: Loop order in main loop

```
#pragma acc parallel loop reduction(max:error)
```

```
for (int iy = iy_start; iy < iy_end; iy++) {
```

```
    #pragma acc loop vector
```

```
    for (int ix = ix_start; ix < ix_end; ix++) {
```

```
        Anew[iy*nx + ix] =
```

```
        ↪ (rhs[iy*nx + ix] +
```

```
            ( A[iy*nx + ix] +
```

```
              + A[(iy-1)*nx + ix] +
```

```
              //...
```

ix Outer loop index; accesses

memory locations

ix; accesses offset

locations

change order to optimize pattern ✓

More on OpenACC thread  
configuration in Appendix!

```
$ make run
```

```
[...]
```

```
2048x2048: Ref: 20.3076 s, This: 0.2602 s, speedup: 78.04
```

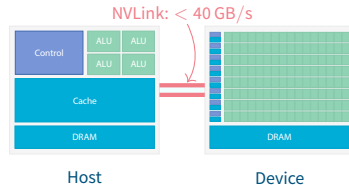
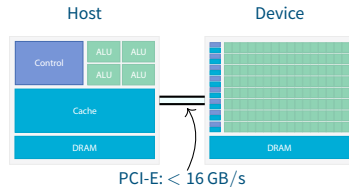
# Aside: Data Transfer with NVLink

- One feature of Minsky not showcased in tutorial: NVLink between CPU and GPU
- Task 3 on P100 + PCI-E:

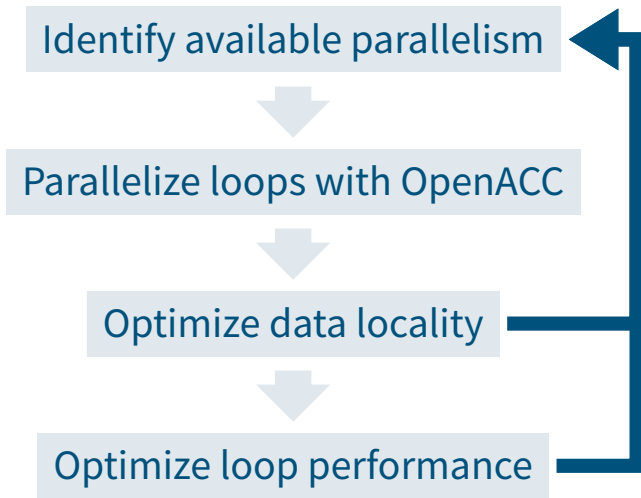
```
$ nvprof ./poisson2d
2048x2048: Ref: 73.1076 s, This: 0.4600 s, speedup: 158.93
Device "Tesla P100-PCI-E-12GB (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    657   149.63KB  4.0000KB  0.9844MB  96.00000MB  9.050452ms  Host To Device
    193   169.78KB  4.0000KB  0.9961MB  32.00000MB  2.679974ms  Device To Host
```

- Task 3 on P100 + NVLink:

```
2048x2048: Ref: 49.7252 s, This: 0.5574 s, speedup: 89.21
Device "Tesla P100-SXM2-16GB (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    480   204.80KB  64.000KB  960.00KB  96.00000MB  3.325184ms  Host To Device
    160   204.80KB  64.000KB  960.00KB  32.00000MB  1.102954ms  Device To Host
```

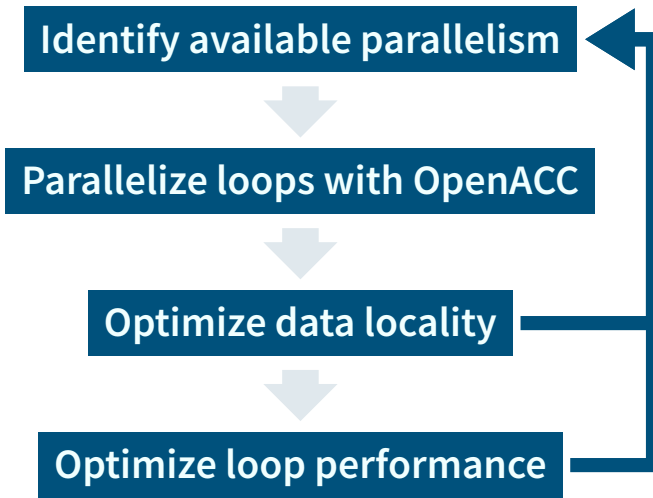


# Parallelization Workflow





# Parallelization Workflow

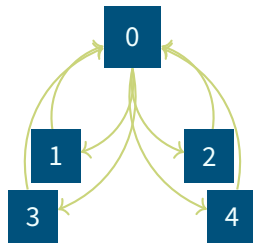


# OpenACC on Multiple GPUs

# Message Passing Interface Introduction

- **MPI:** Message Passing Interface
- Standardized API to communicate data across processes and nodes; compilers
- Various implementations: OpenMPI, MPICH, MVAPICH, Vendor-specific versions
- Standard in parallel and distributed High Performance Computing
- Unrelated to OpenACC, but works well together!

→ [www.open-mpi.org/doc/](http://www.open-mpi.org/doc/)



# MPI API Examples

- Configuration calls

`MPI_Comm_size()` Get number of total processes

`MPI_Comm_rank()` Get current process number



- Point-to-point routines

`MPI_Send()` Send data to other process

`MPI_Recv()` Receive data from other process

`MPI_Sendrecv()` Do both in one call

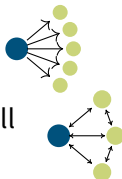


- Collective routines

`MPI_Bcast()` Broadcast data from one process to all others

`MPI_Reduce()` Reduce (e.g. sum) values on all processes

`MPI_Allgather()` Gathers data from all processes, distributes to all



- *And many, many more!*

# MPI Skeleton

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    // Initialize MPI
    MPI_Init(&argc, &argv);

    int rank, size;
    // Get current rank ID
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Get total number of ranks
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Do something (call MPI routines, ...)
    ...

    // Shutdown MPI
    MPI_Finalize();
    return 0;
}
```

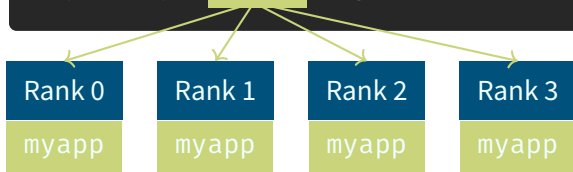
# Using MPI

- Compile with MPI compiler (wrapper around usual compiler)

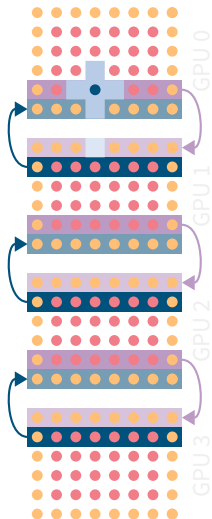
```
$ mpicc -o myapp myapp.c
```

- Run with MPI launcher `mpirun` (takes care about configuration, `$VARS`, ...)

```
$ mpirun -np 4 ./myapp <arguments>
```



# MPI Strategy for Jacobi Solver



- Goal: Extend parallelization from GPU threads to multiple GPUs
- Distribute grid of points to GPUs
- Halo points ● need special consideration

*That's what makes things interesting here*

- Evaluated point ● needs data from neighboring points ■
- At border: Data might be on different GPU → Halos! ■
- For every iteration step: Update halo from other GPU device  
⇒ Regular MPI communications to top ■ and from top ■

```
MPI_Sendrecv(A+iy_start*nx+1, nx-2, MPI_DOUBLE, top, 0,  
             A+iy_end*nx+1, nx-2, MPI_DOUBLE, bottom, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(A+(iy_end-1)*nx+1, nx-2, MPI_DOUBLE, top, 0,  
             A+(iy_start-1)*nx+1, nx-2, MPI_DOUBLE, bottom, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

# Determining GPU ID

## Affinity on nodes with multiple GPUs

- Problem: Usually, nodes have more than one GPU
- How would MPI know how to distribute the load?
- Select active GPU with  
`#pragma acc set device_num(ID)`
- Alternative and more [in appendix](#)



# Parallel Jacobi III

## Multi-GPU parallelism, asynchronous execution

TASK 4

- Implement domain decomposition for 4 GPUs

### Task 4: Multi-GPU Usage

- Change to Task4/ directory
- Change source code; see README.md
- Compile: make
- Submit parallel run to the batch system: make run

? What's your speed-up?

**E** Implement asynchronous halo communication; see README.md in Task4E/!

# Parallel Jacobi III

## Source Code

```
#pragma acc set device_num(rank)
// ...
int iy_start = rank * chunk_size;
int iy_end   = iy_start + chunk_size;
// ...
MPI_Sendrecv( A+iy_start*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top, 0,
              A+iy_end*nx+ix_start,   (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
MPI_Sendrecv( A+(iy_end-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
              A+(iy_start-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
```

# Parallel Jacobi III

## MPI run result

```
$ make run
$ bsub -env "all" -n 4 -I -R "rusage[ngpus_shared=1]" mpirun --npersocket 2 -bind-to core
-np 4 ./poisson2d 1000 4096
Job <15145> is submitted to queue <vis>.
Jacobi relaxation calculation: max 1000 iterations on 4096 x 4096 mesh
Calculate reference solution and time with MPI-less 1 device execution.
    0, 0.250000
   100, 0.249940
   [...]
Calculate current execution.
    0, 0.250000
   [...]
Num GPUs: 4.
4096x4096: 1 GPU:   1.8621 s, 4 GPUs:   0.6924 s, speedup:      2.69, efficiency:    67.23%
MPI time:   0.1587 s, inter GPU BW:   0.77 GiB/s
```

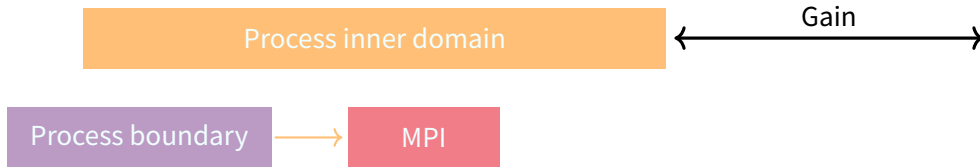
# Overlap Communication and Computation

## Disentangling

No Overlap



Overlap



# Overlap Communication and Computation

## OpenACC keyword

- OpenACC: Enable asynchronous execution with `async` keyword
- Runtime will execute `async`'ed region at same time
- Barrier: `wait`

```
#pragma acc parallel loop present(A, Anew)
for ( ... ) { } // Process boundary
#pragma acc parallel loop present(A, Anew) async
for ( ... ) { } // Process inner domain
#pragma acc host_data use_device (A) {
    MPI_Sendrecv(A+iy_start*nx+1, nx-2, MPI_DOUBLE, top, 0,
                 A+iy_end*nx+1, nx-2, MPI_DOUBLE, bottom, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(A+(iy_end-1)*nx+1, nx-2, MPI_DOUBLE, bottom, 1,
                 A+(iy_start-1)*nx+1, nx-2, MPI_DOUBLE, top, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
#pragma acc wait // Wait for inner domain to finish processing
```

# Parallel Jacobi III+

E

## MPI async run result

```
$ make run
$ bsub -env "all" -n 4 -I -R "rusage[ngpus_shared=1]" mpirun --npersocket 2 -bind-to core
  -np 4 ./poisson2d 1000 4096
Job <15145> is submitted to queue <vis>.
Jacobi relaxation calculation: max 1000 iterations on 4096 x 4096 mesh
Calculate reference solution and time with MPI-less 1 device execution.
    0, 0.250000
   100, 0.249940
   [...]
Calculate current execution.
    0, 0.250000
   [...]
Num GPUs: 4.
4096x4096: 1 GPU:   1.8656 s, 4 GPUs:   0.6424 s, speedup:      2.90, efficiency:    72.61%
MPI time:   0.2455 s, inter GPU BW:   0.50 GiB/s
```

# Conclusions, Summary

# Conclusions & Summary

We've learned a lot today!

- **Minsky** nodes are fat nodes:  
2 POWER8NVL CPUs ( $2 \times 10$  cores), 4 P100 GPUs ( $4 \times 56$  SMs)
  - **OpenACC** can be used to efficiently exploit parallelism
  - ... on the CPU, similar to OpenMP,
  - ... on the GPU, for which it is specially designed for,
  - ... on multiple GPUs, working well together with MPI.
  - There are still many more tuning possibilities and keywords (see a ...)
- Great online resources to **deepen your knowledge** (see a ...)

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



# APPENDIX

## Appendix

List of Tasks

Supplemental: POWER9 Structure Diagrams

Supplemental: JURON Login via SSH

Supplemental: Summitdev Login

Supplemental: NVIDIA GPU Memory Spaces

Supplemental: Leveraging OpenACC Threads

Supplemental: MPI

Further Reading

Glossary

References

# List of Tasks

Task 1: JURON

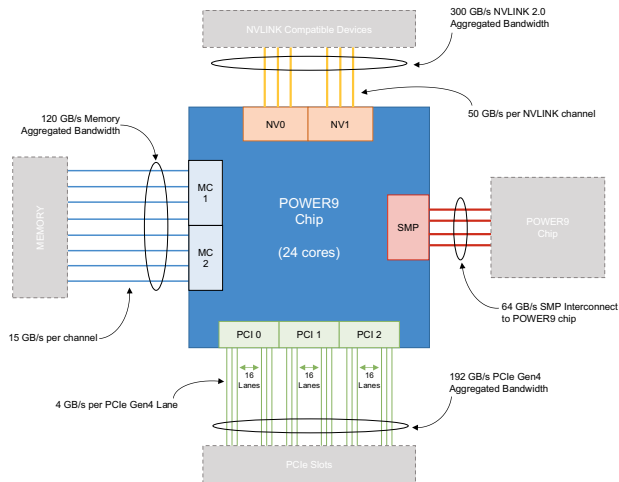
Task 2: A First Parallel Loop

Task 3: More Parallel Loops

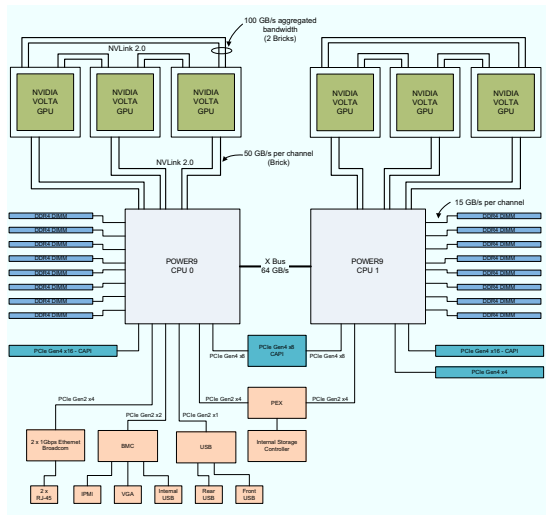
Task 4: Multi-GPU Usage

## Supplemental: POWER9 Structure Diagrams

# POWER9 Structure Diagram



# Newell Structure Diagram



## Supplemental: JURON Login via SSH

# JURON Login via SSH

- Download SSH key from <http://bit.ly/gtc18-openacc>; OpenSSH (Linux, Mac, Windows) and PuTTY (Windows) keys provided

*Set right permissions to key: `chmod 600 mykey`*

- Unlock key with password from slip of paper

- Log in to JURON

```
ssh -i id_train0XX train0XX@juron.fz-juelich.de
```

- Prevent entering passphrase multiple times: Add key to SSH agent

```
eval "$(ssh-agent -s)"
```

```
ssh-add train0XX
```



## Supplemental: Summitdev Login

# Using Summitdev

- Summitdev: Access via RSA tokens
- Login first to `home.ccs.ornl.gov` then to summitdev ([docs](#))
  - First** Connect with key on RSA token; set PIN (4-6 digits); confirm with PIN followed by RSA Passphrase
  - All other** Connect with PIN followed by RSA Passphrase
- Checkout Lab repository

```
git clone -b summitdev https://gitlab.version.fz-juelich.de/herten1/gtc18-openacc.git
```
- Load required modules

```
module load pgi/18.1 cuda
```
- Allocate resources on compute nodes

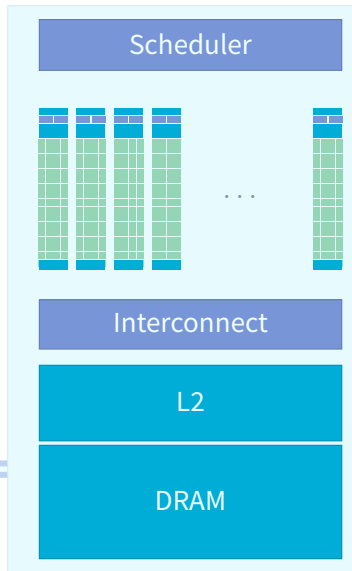
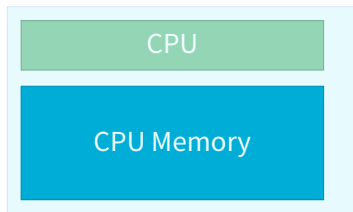
```
bsub -nnodes 1 -W 120 -P "TRN001" -Is SHELL
```
- Run jobs: `jsrun -n1 [...] ./prog` ([docs](#))

## Supplemental: NVIDIA GPU Memory Spaces

# NVIDIA GPU Memory Spaces

Location, location, location

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



# NVIDIA GPU Memory Spaces

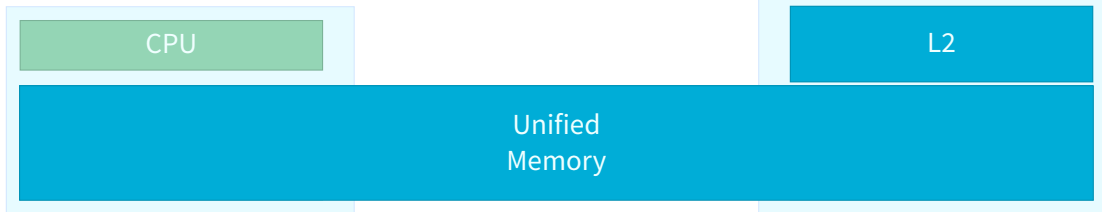
## Location, location, location

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)



## Supplemental: Leveraging OpenACC Threads

# Understanding Compiler Output

```
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] + A[iy*nx+ix-1] +
117              ↪ 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  }
```

# Understanding Compiler Output

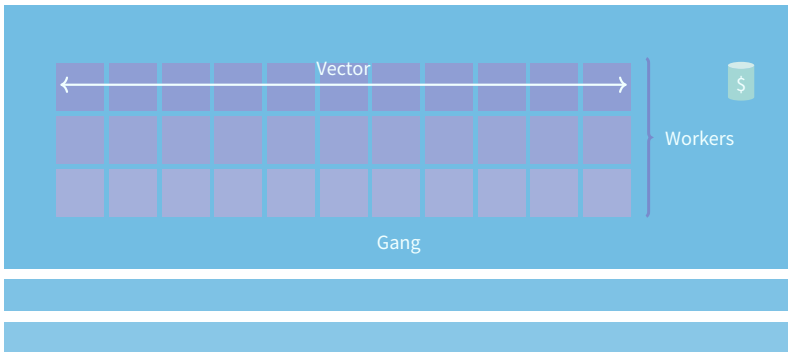
```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

- 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

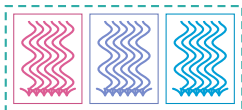
### Software



Thread



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`  $\times$  `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)

# Understanding Compiler Output II

```
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

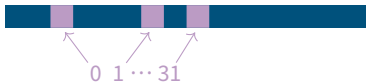
## Compiler Output

```
$ 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 */
    ...
```

# Memory Coalescing

## Memory in batch

- 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



## Supplemental: MPI



# Handling Multi-GPU Hosts

## The alternative

- Use OpenACC API to select GPU

```
#if _OPENACC  
acc_device_t device_type = acc_get_device_type(); // Get dev type  
int ngpus = acc_get_num_devices(device_type); // Get number of devs  
int devicenum = rank%ngpus; // Compute active dev number based on rank  
acc_set_device_num(devicenum, device_type);  
#endif /*_OPENACC*/
```

- Get rank ID

- MPI API: MPI\_Comm\_rank()
- Environment variables (int rank = atoi(getenv(...)))

```
OpenMPI $OMPI_COMM_WORLD_LOCAL_RANK  
MVAPICH2 $MV2_COMM_WORLD_LOCAL_RANK
```

## Further Reading

## Further Resources on OpenACC

- [www.openacc.org](http://www.openacc.org): Official home page of OpenACC
- [developer.nvidia.com/openacc-courses](http://developer.nvidia.com/openacc-courses): OpenACC courses, upcoming (live) and past (recorded)
- <https://nvidia.qwiklab.com/quests/3>: Qwiklabs for OpenACC; various levels
- Book: **Chandrasekaran and Juckeland** *OpenACC for Programmers: Concepts and Strategies* <https://www.amazon.com/OpenACC-Programmers-Strategies-Sunita-Chandrasekaran/dp/0134694287> [11]
- Book: **Farber** *Parallel Programming with OpenACC* <https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979> [12]

# Glossary I

**API** A programmatic interface to software by well-defined functions. Short for application programming interface. [75](#), [76](#), [113](#)

**CUDA** Computing platform for [GPUs](#) from NVIDIA. Provides, among others, CUDA C/C++. [25](#), [100](#), [101](#), [107](#)

**GCC** The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. [24](#), [27](#)

**JULIA** One of the two HBP pilot system in Jülich; name derived from Juelich and Glia. [9](#)

**JURON** One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. [3](#), [4](#), [9](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#)

# Glossary II

**MPI** The Message Passing Interface, a API definition for multi-node computing. 75, 76, 77, 78, 79, 80, 83, 86, 88, 90, 112, 113

**NVIDIA** US technology company creating GPUs. 4, 5, 6, 20, 90, 99, 100, 101, 116

**NVLink** NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 5, 7, 8, 10, 71, 116

**OpenACC** Directive-based programming, primarily for many-core machines. 2, 3, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 33, 38, 39, 41, 43, 45, 50, 54, 55, 57, 58, 61, 62, 66, 67, 68, 69, 70, 72, 73, 75, 85, 88, 90, 102, 105, 107, 108, 113, 115

**OpenMP** Directive-based programming, primarily for multi-threaded machines. 21, 45, 49, 52, 88

# Glossary III

**P100** A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast HBM2 memory. 6, 7, 8, 71, 88

**PAPI** The Performance API, a C/C++ API for querying performance counters. 34

**Pascal** GPU architecture from NVIDIA (announced 2016). 10, 100, 101, 116

**perf** Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. 34

**PGI** Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 24, 27, 34, 49

**POWER** CPU architecture from IBM, earlier: PowerPC. See also POWER8. 2, 3, 4, 5, 7, 8, 10, 90, 92, 93, 116

# Glossary IV

**POWER8** Version 8 of IBM's **POWER** processor, available also under the OpenPOWER Foundation. [10](#), [88](#), [116](#)

**Tesla** The **GPU** product line for general purpose computing computing of **NVIDIA**. [6](#)

**Volta** **GPU** architecture from **NVIDIA** (announced 2017). [10](#)

**CPU** Central Processing Unit. [2](#), [3](#), [5](#), [6](#), [7](#), [8](#), [10](#), [24](#), [45](#), [56](#), [69](#), [71](#), [88](#), [100](#), [101](#), [116](#)

**GPU** Graphics Processing Unit. [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [19](#), [24](#), [26](#), [54](#), [56](#), [57](#), [59](#), [71](#), [79](#), [80](#), [81](#), [88](#), [90](#), [99](#), [100](#), [101](#), [113](#), [116](#)

**HBP** Human Brain Project. [9](#), [116](#)

# Glossary V

**SM** Streaming Multiprocessor. 7, 10, 88

**SMT** Simultaneous Multithreading. 7



# References I

- [7] The Next Platform. *Power9 To The People*. POWER9 Performance Data. URL: <https://www.nextplatform.com/2017/12/05/power9-to-the-people/>.
- [8] Alexandre Bicas Caldeira. *IBM Power System AC922: Introduction and Technical Overview*. IBM Redbooks. URL: <http://www.redbooks.ibm.com/redpieces/pdfs/redp5472.pdf> (pages 93, 94).
- [10] Donald E. Knuth. “Structured Programming with Go to Statements”. In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: [10.1145/356635.356640](https://doi.org/10.1145/356635.356640). URL: <http://doi.acm.org/10.1145/356635.356640> (page 34).

## References II

- [11] Sunita Chandrasekaran and Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017. ISBN: 0134694287. URL: <https://www.amazon.com/OpenACC-Programmers-Strategies-Sunita-Chandrasekaran/dp/0134694287> (page 115).
- [12] Rob Farber. *Parallel Programming with OpenACC*. Morgan Kaufmann, 2016. ISBN: 0124103979. URL: <https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979> (page 115).

# References: Images, Graphics I

- [1] SpaceX. *SpaceX Launch*. Freely available at Unsplash. URL: <https://unsplash.com/photos/uj3hvdfQujI>.
- [2] Forschungszentrum Jülich. *Hightech made in 1960: A view into the control room of DIDO*. URL: [http://historie.fz-juelich.de/60jahre/DE/Geschichte/1956-1960/Dekade/\\_node.html](http://historie.fz-juelich.de/60jahre/DE/Geschichte/1956-1960/Dekade/_node.html) (page 4).
- [3] Forschungszentrum Jülich. *Forschungszentrum Bird's Eye*. (Page 4).
- [4] Forschungszentrum Jülich. *JUQUEEN Supercomputer*. URL: [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN\\_node.html](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html) (page 4).

## References: Images, Graphics II

- [5] Rob984 via Wikimedia Commons. *Europe orthographic Caucasus Urals boundary (with borders)*. URL: [https://commons.wikimedia.org/wiki/File:Europe\\_orthographic\\_Caucasus\\_Urals\\_boundary\\_\(with\\_borders\).svg](https://commons.wikimedia.org/wiki/File:Europe_orthographic_Caucasus_Urals_boundary_(with_borders).svg) (page 4).
- [6] IBM AIXpert Blog. *IBM Minsky Picture*. URL: [https://www.ibm.com/developerworks/community/blogs/aixpert/entry/OpenPOWER\\_IBM\\_S822LC\\_for\\_HPC\\_Minsky\\_First\\_Look?lang=en](https://www.ibm.com/developerworks/community/blogs/aixpert/entry/OpenPOWER_IBM_S822LC_for_HPC_Minsky_First_Look?lang=en) (page 6).
- [9] Setyo Ari Wibowo. *Ask*. URL: <https://thenounproject.com/term/ask/1221810>.