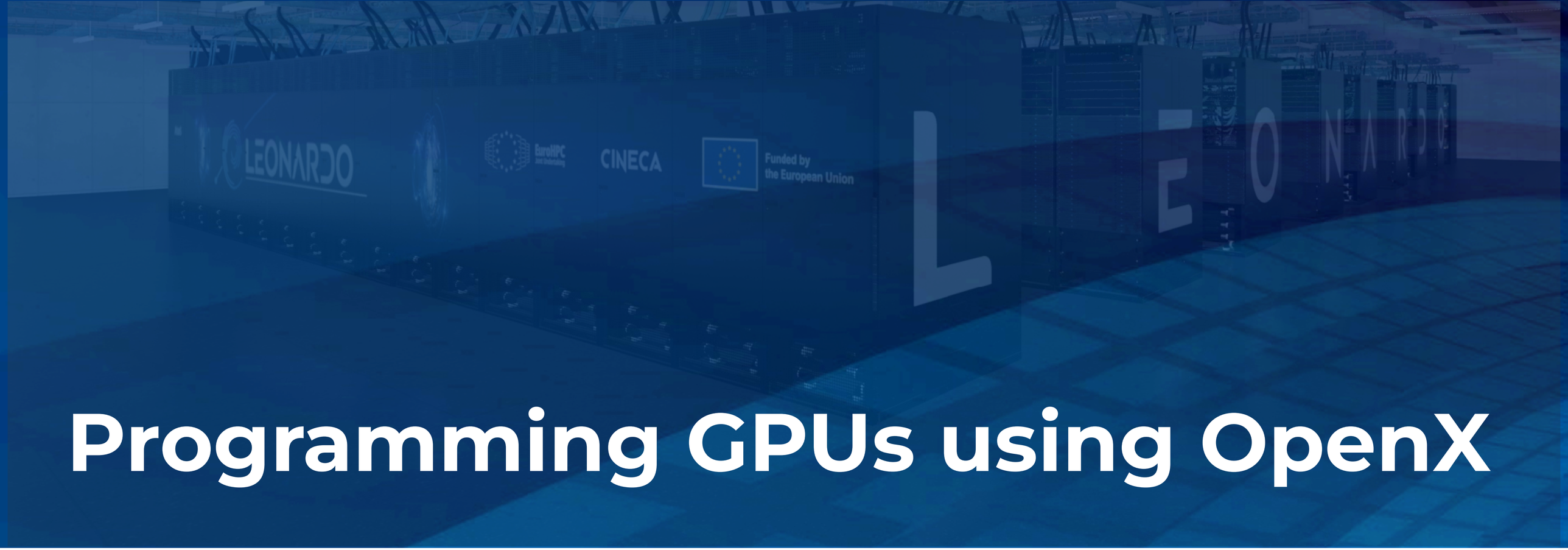


Programming GPUs using OpenX

N. Shukla

CINECA HPC Summer School at Bologna Italy

Email: n.shukla@cineca.it

July 9-10th, 2024

Objectives

What are Compiler Directives?

Easy way porting code to GPU

What is OpenACC?

Execution Model

Data Locality

Memory model

Case study

House keeping

- Git clone
- cd Day-6th

```
salloc -N1 --cpus-per-task=1 --ntasks-per-node=8 -A XX -t 00:00:00 -p
boost_usr_prod —qos=XX --gres=gpu:1
```

Asking Interactive node

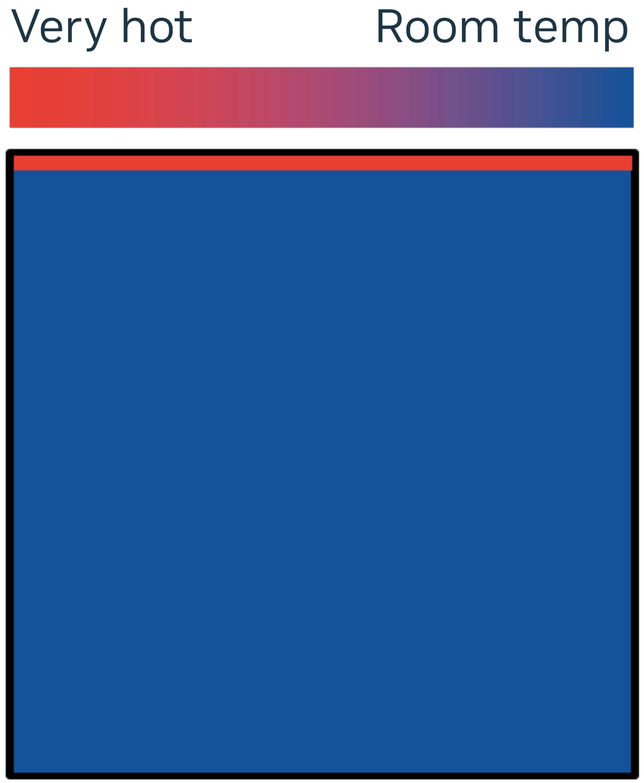
Case study: Parallelise a serial Laplace 2D

Laplace heat transfer

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.

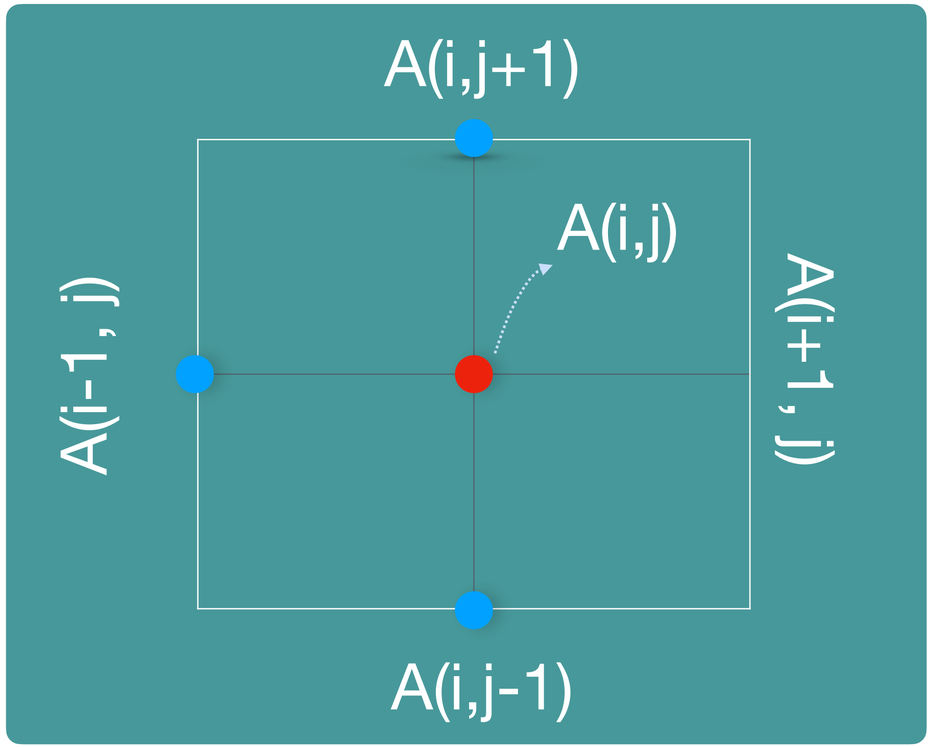


Code description

- Iteratively converges to correct value (e.g. Temperature)
- by computing new values at each point from the average of neighboring points.
- Example: Solve Laplace equation in 2D

$$\nabla^2 f(x,y) = 0$$

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



Code description

```
while (error > tol && niter < niter_max)
error = 0.0;

 for (int j = 1; j < n-1; ++j) {
  for (int i = 1; i < m-1; ++i) {
   Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m]
   + A[idx+m]);

   error = fmax(error, fabs(Anew[idx] - A[idx])); }
   }
 for (int j = 1; j < n-1; ++j)
  for (int i = 1; i < m-1; ++i)
   A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

Compute max error for converge

Compute max error for converge

Swap input/output arrays

Compiling options

NVIDIA's HPC Compilers (AKS PGI)

Compiling sequential code

NVIDIA compiler Names (PGI may still work)

- nvc - The command to compiler C code (pgcc)
- nvc++ - The command to compiler C++ code (pgc++)
- nvfortran - The command to compiler Fortran code (pgfortran)
- The -fast flag instructs the compiler to optimise the code to the best of its abilities

```
nvc -fast my_program.c
nvc++ -fast my_program.cpp
nvfortran -fast my_program.cf90
```

NVIDIA's HPC Compilers (AKS PGI)

-Minfo flag

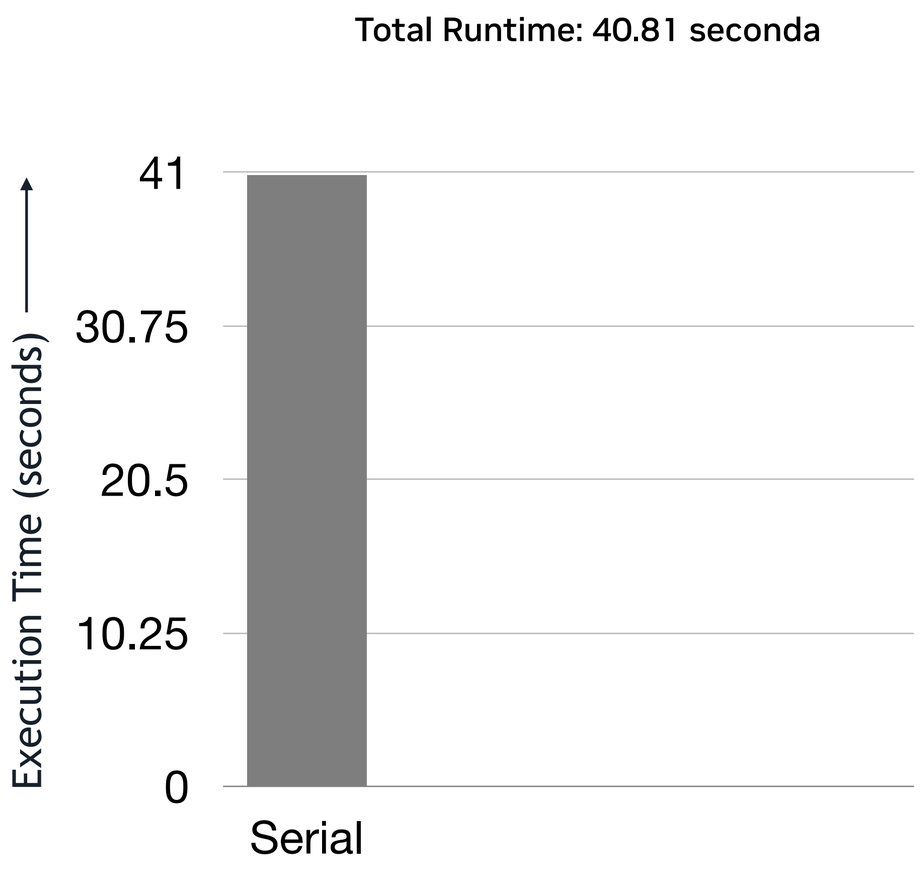
Instruct compiler to print feedback about the compiled code

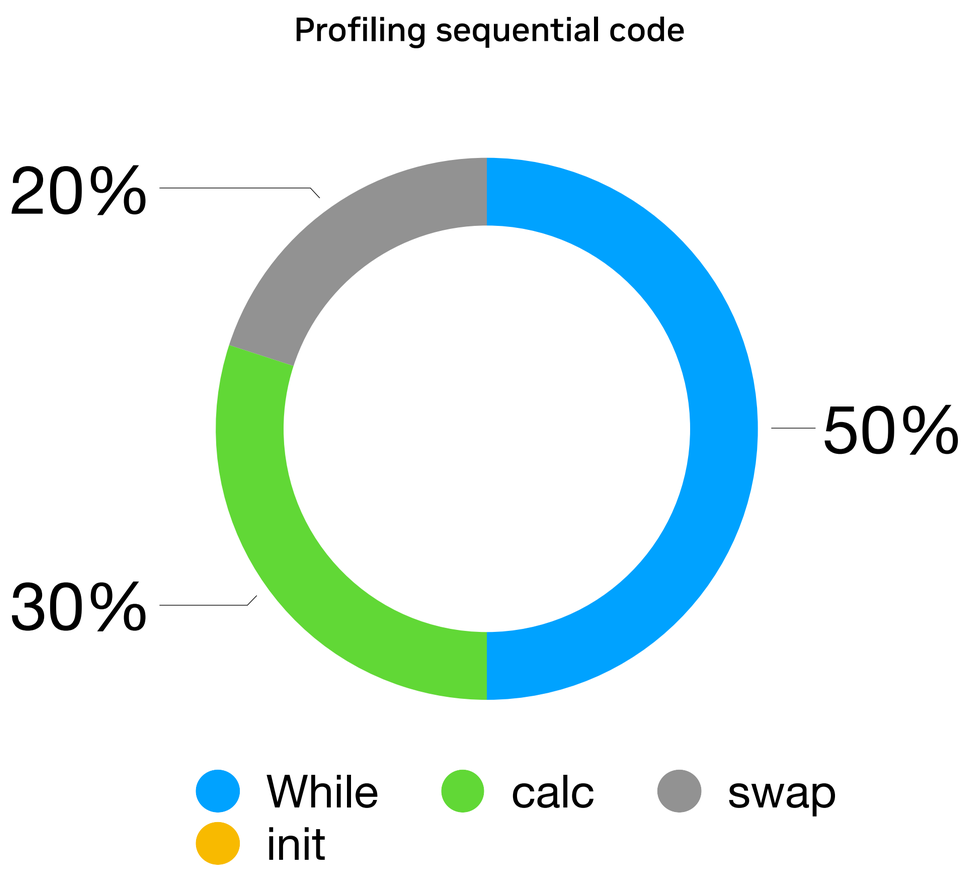
- -Minfo=all gives all code feedback, whether positive or negative
- --Minfo = opt informs about all code optimisations

```
nvc -fast -Minfo=all my_program.c
nvc++ -fast -Minfo=all my_program.cpp
nvfortran -fast -Minfo=all  my_program.cf90
```

Hotspots: Identify the portions of code that took the longest to run

Simulation was performed 1000 Iterations



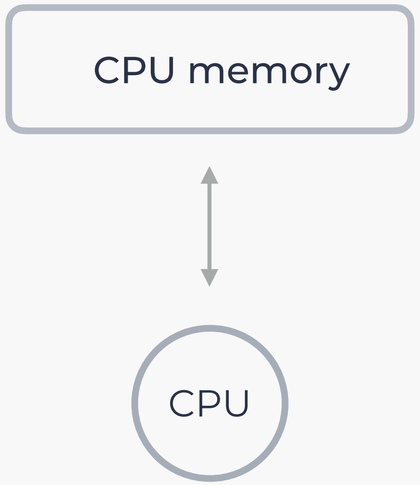


CINECA

Why GPUS?

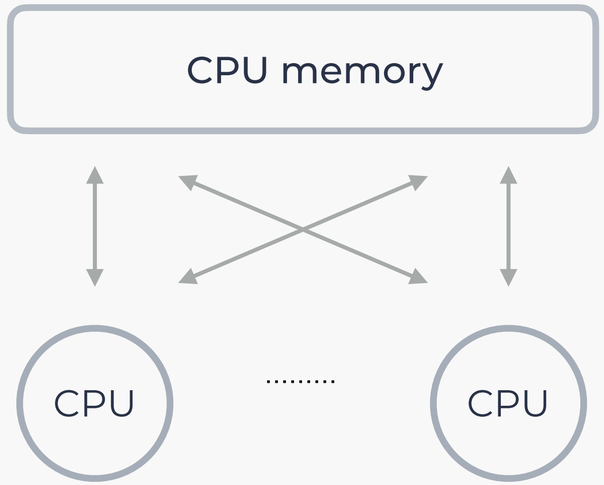
HPC System Evolution: Silicon level parallelism

Changes in the architecture level also requires changes in the programming models



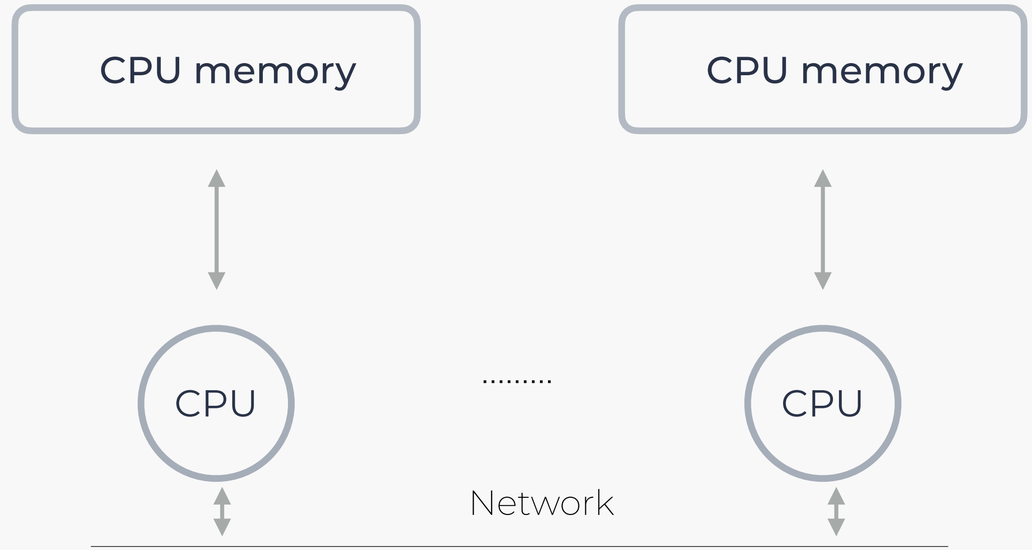
Single CPU

- Sequential computing
- Performance:Mflop/sec



Multiple CPU

- Shared memory
- Required in change hardware and software architectures
- New standards: pThreads or OpenMP
- Performance: Gflops/sec



Multiple-node system

- Connected with high-speed and low latency network (200 GB/s via Infi band network)
- Requires change in hardware and software architecture
- New standard: MPI+OpenMP
- Performance: Tflop/sec

Golden era: 1977-2017

40 years of stunning progress in microprocessor design

1.4x annual performance improvement for 40+ years ~ 106x faster (throughput)

Three architectural innovations

Width 8 > 16 > 64 bit (~ 4x)
Instruction level parallelisms: 4-10 cycles per instructions to 4+instructions per cycle (~10-20x)
Multicore: one processor to 32 processor (~32x)

Through technology and architecture

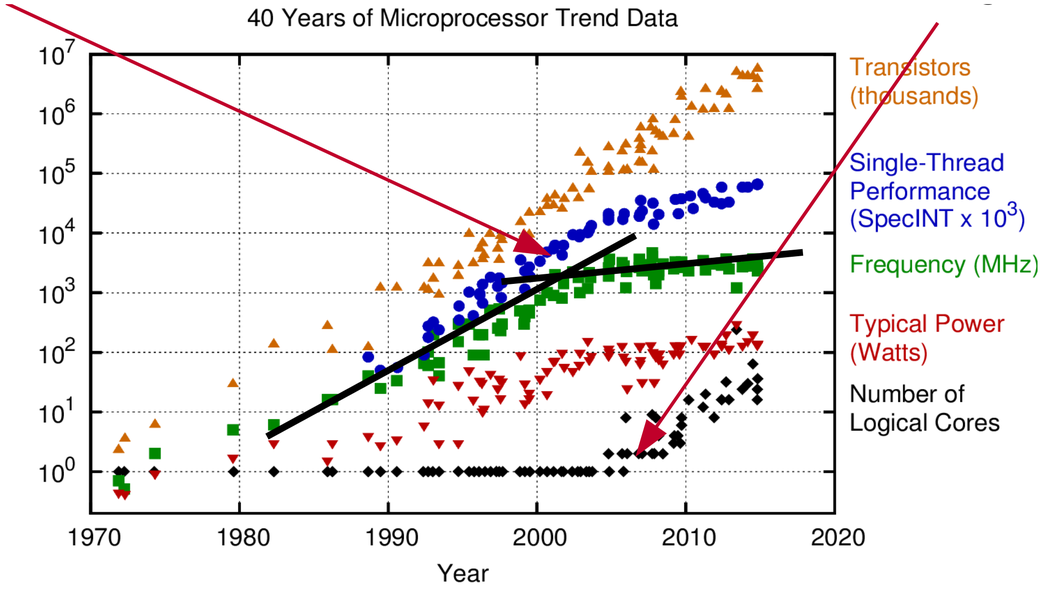
Clock rate: 3MHZ to 4GHZ

Made possible by IC technology

W Moore's law: growth in transistor count
Dennard scaling: power/transistor shrinks at same rate as transistors are added (constant per mm2 of silicon), held until 1997 and then began fade away
2007-2017: 45 to 16nm : 3.0x increase in energy/chip

Clock speed stopped increasing due to heat limit

Multiple core processors emerge (Intel i7: 4 cores)

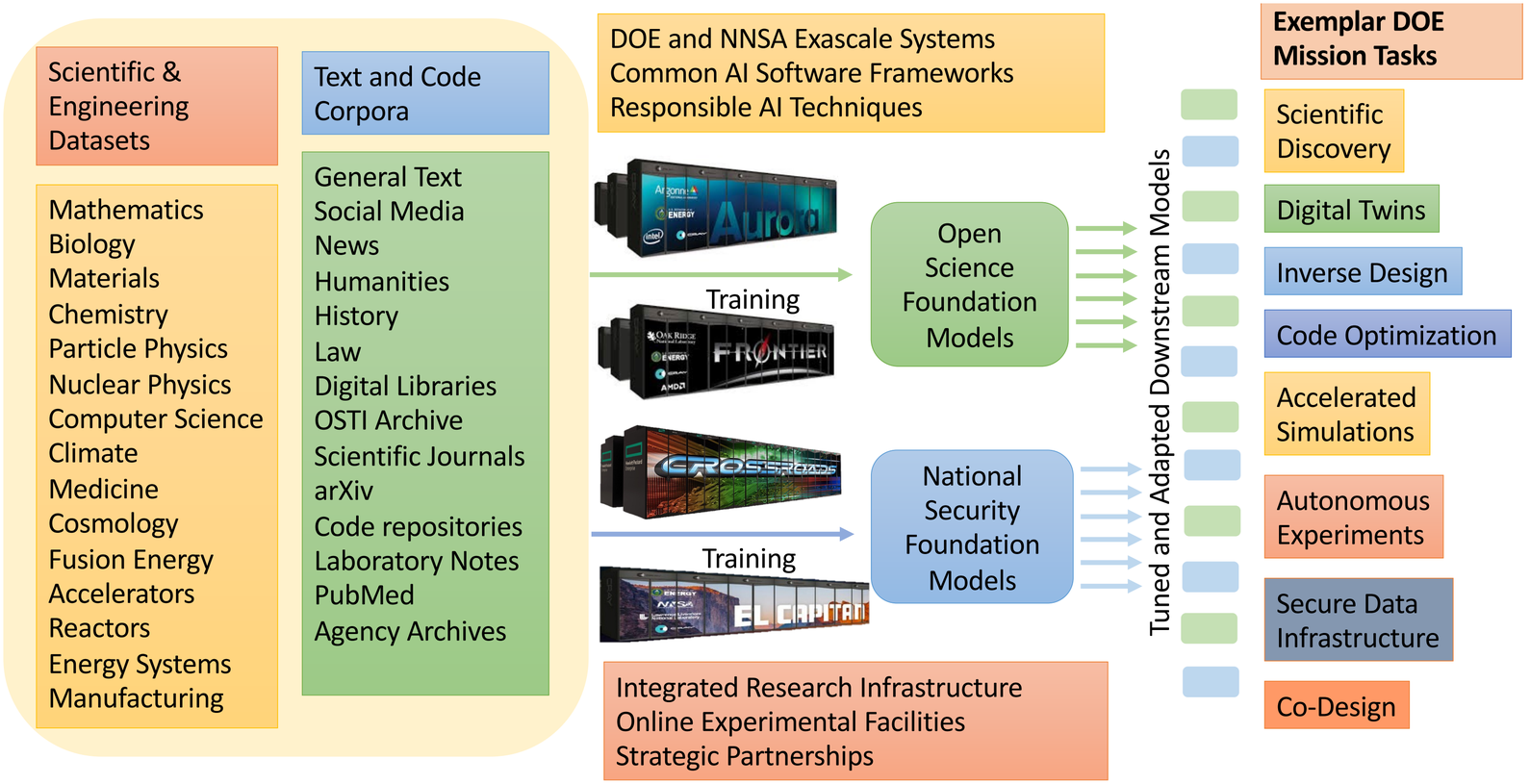


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Delivering on Exascale Science

$\Longrightarrow$ Large Number of Applications

When performance matters



These powerful HPC clusters are Power hungry

But what’s stop building conventional clusters?

Frontier ORNL delivers about 20 % of the entire list's compute performance

Computer System requirements for each Leadership Computing Center

	2012	2018	2022	2026
Peak FLOP/s	10-20 PF	100-200 PF	500-2000 PF	2000-4000 PF
Memory	0.5-1 PB	5-10 PB	32-64 PB	50-100 PB
I/O Buffer	N/A	500 TB	3 PB	5 PB
Storage Disk +tape	20+100 PB	100+1000 PB	1+10 EB	5+50 EB
Power & Space	6-12 MW 5,000-10,000 ft^2	15-20 MW 8,000-15,000 ft^2	20-30 MW 20,000 ft^2	25-35 MW 25,000 ft^2

HPC's Biggest Challenge: Power

TOP10 System - November 2023

1. Frontier ORNL

AMD CPUs
AMD GPUs
HPE Slingshot
1679 pflops

2. Aurora ANL

Intel CPUs
Intel GPUs
HPE Slingshot
1059 pflops

3. Eagle Microsoft

Intel CPUs
Nvidia GPUs
Nvidia Inf
846 pflops

4. Fugaku RIKEN

Fujitsu ARM

Fujitsu Tofu
537 pflops

5. Lumi CSC

AMD CPUs
AMD GPUs
HPE Slingshot
531 pflops

6. Leonardo CINECA

Intel CPUs
Nvidia GPUs
Nvidia Inf
304 pflops

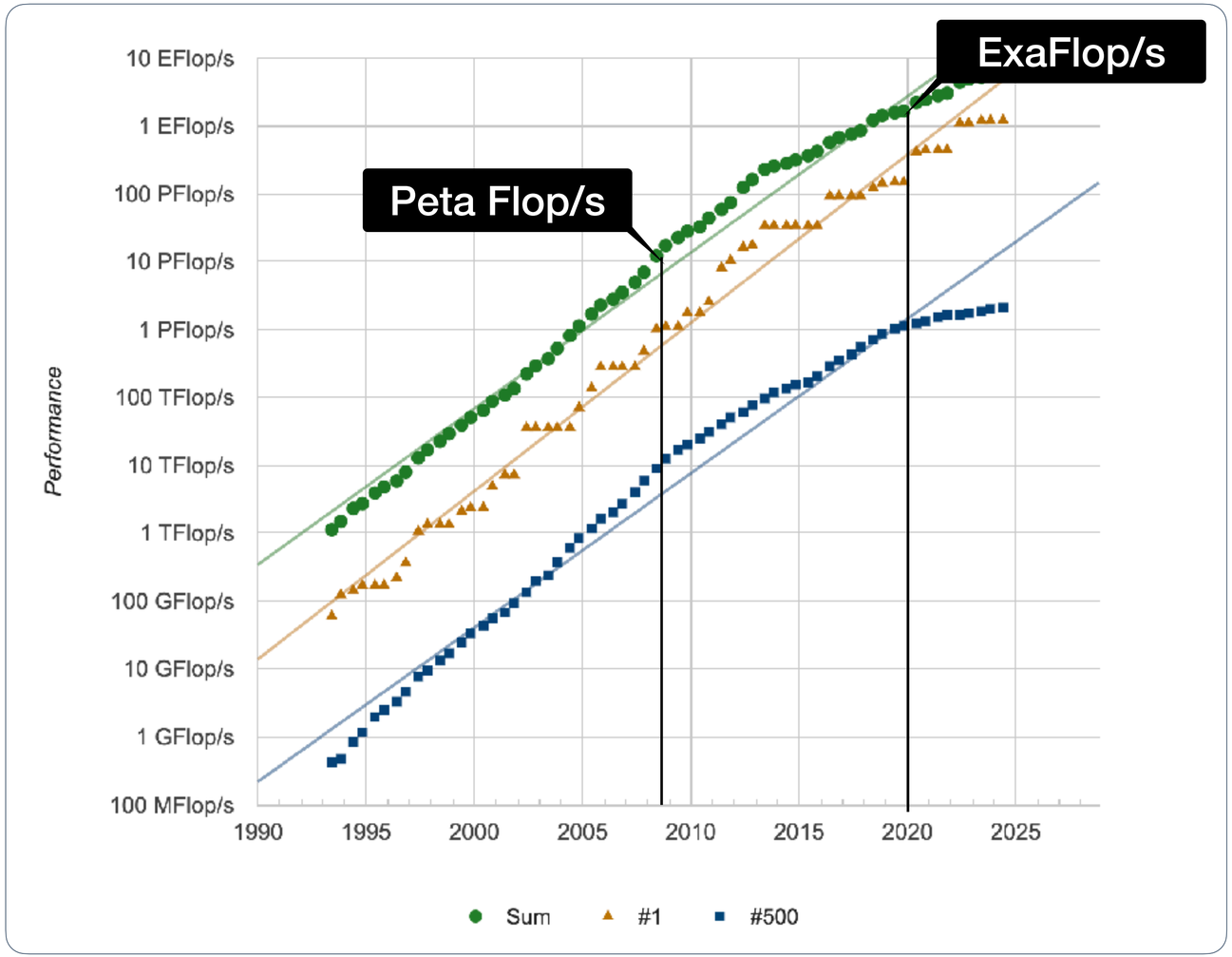
70 % of FLOP/s by GPUs, > 100 000 GPUs in Frontier+Aurora

Why to care about accelerators?

FLOPS: FLoating-point OPerations per Second

1EF = 1,000,000,000,000,000,000

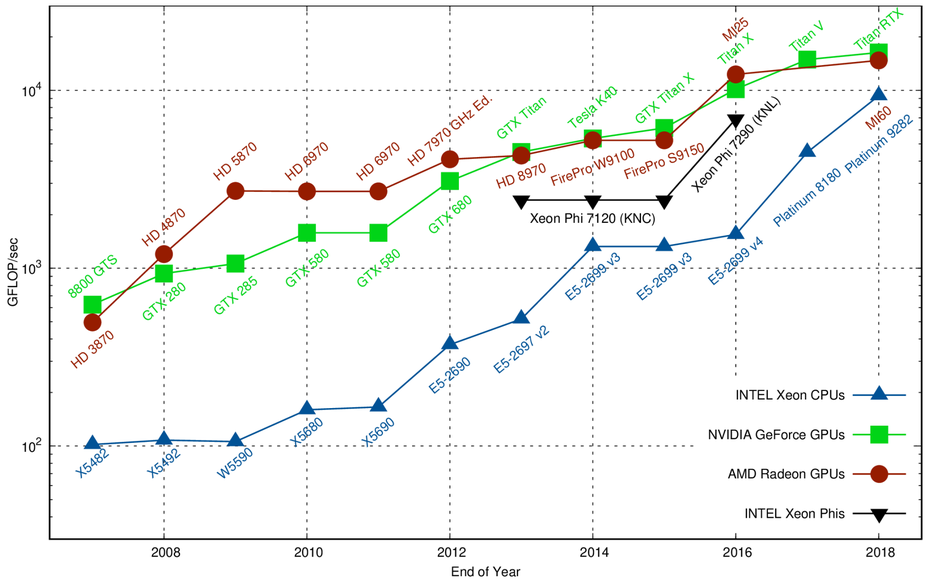
a supercomputer that can calculate at least one quintillion floating point operations per second.



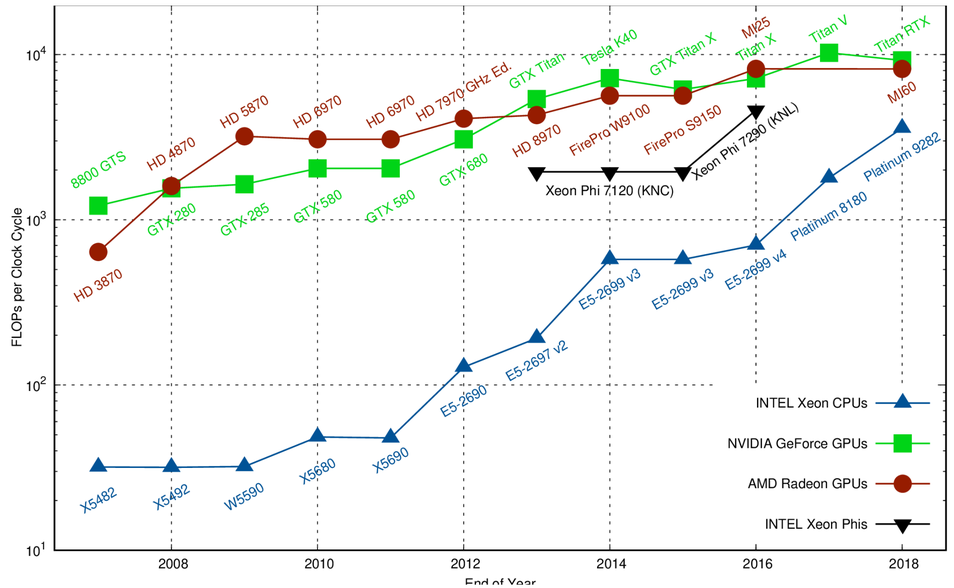
Evolution of peaks FLOPS

- Gaming industry evolves steadily → continuous high demand for consumer GPUs
- With the trend for AI in many different areas → continuous demand for professional GPUs

Theoretical peak performance, single precision

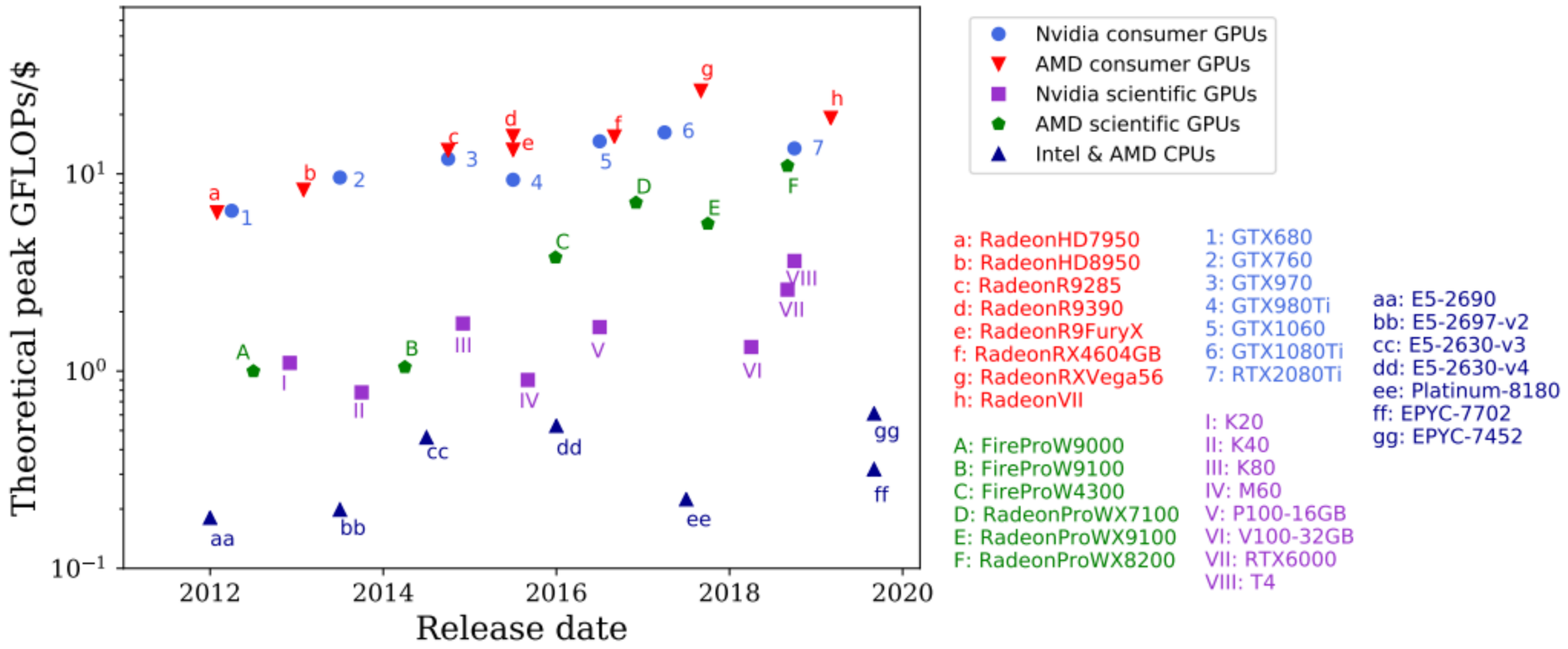


Theoretical peak FLOPs per clock cycle, single precision



CINECA

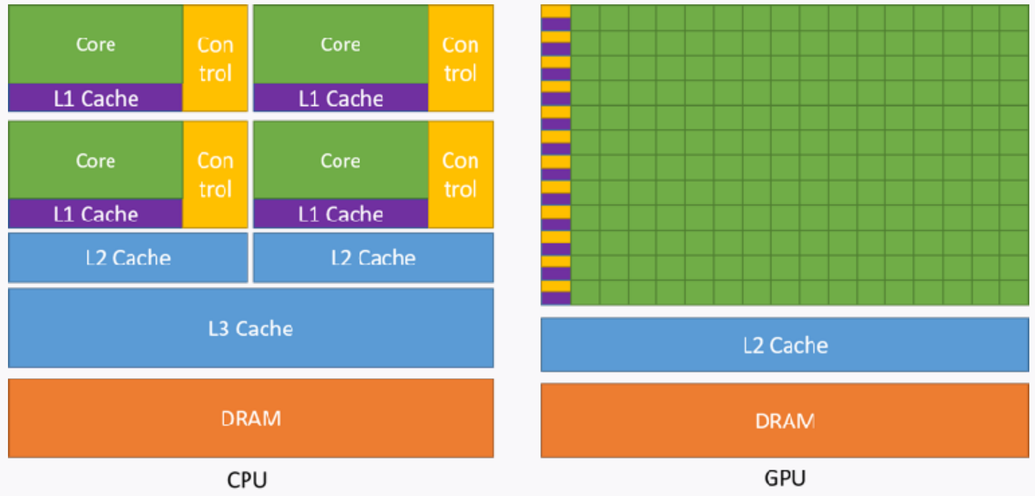
Theoretical FLOPs/$: GPUs & CPU



https://arxiv.org/pdf/2003.11491.pdf

Why can accelerators deliver good performance watt ratio?

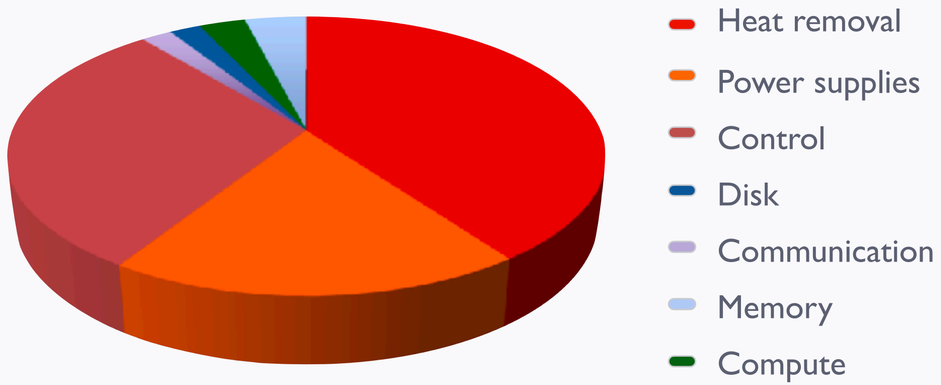
- **High (peak) performance**
  - More transistors for computations
  - No control logic
  - Small caches



- **Low power consumption**
  - Many low frequency cores

    $P \sim V^2 \times f$
  - No control logic

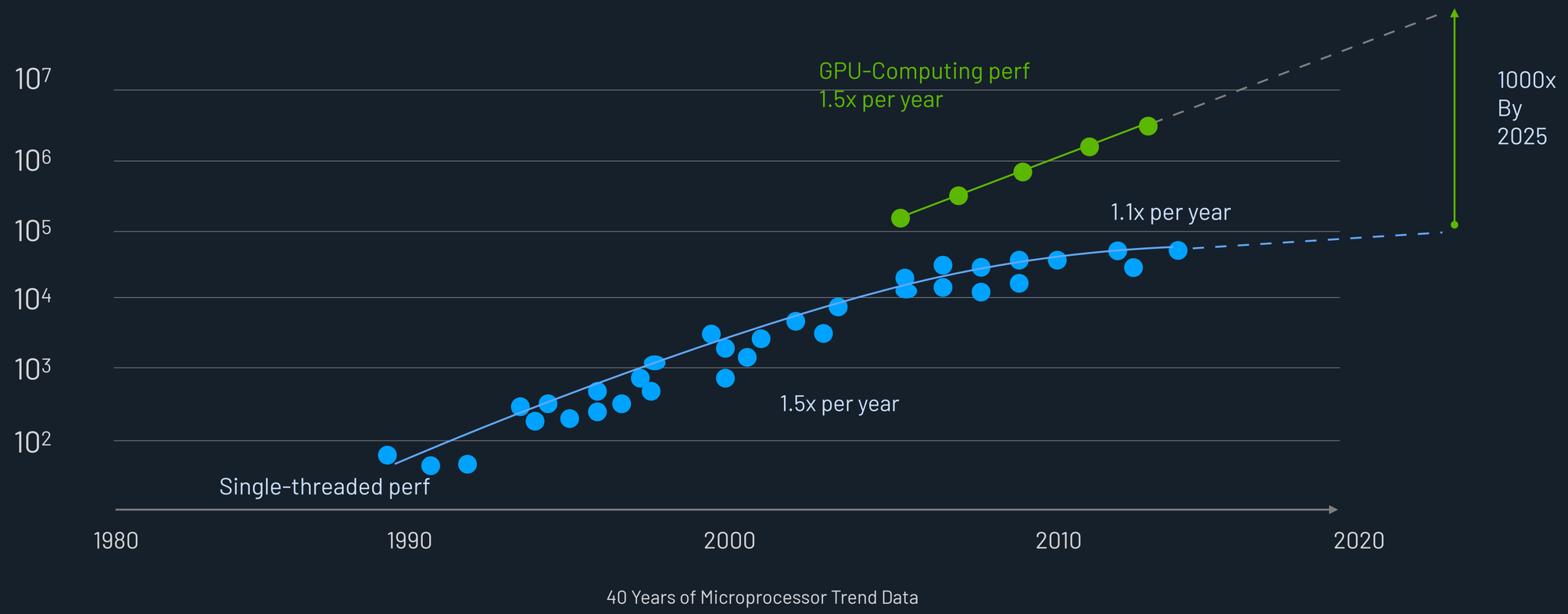
- **Power use for 1 Tflops/s of an usual system**



Source: Andrey Semin (Intel): HPC systems energy efficiency optimization thru hardware-software co-design on Intel technologies, EnaHPC 2011; & S. Borkar, J. Gustafson

Why the GPU computing trend?

GPU exhibits intense computational powers than traditional CPUs

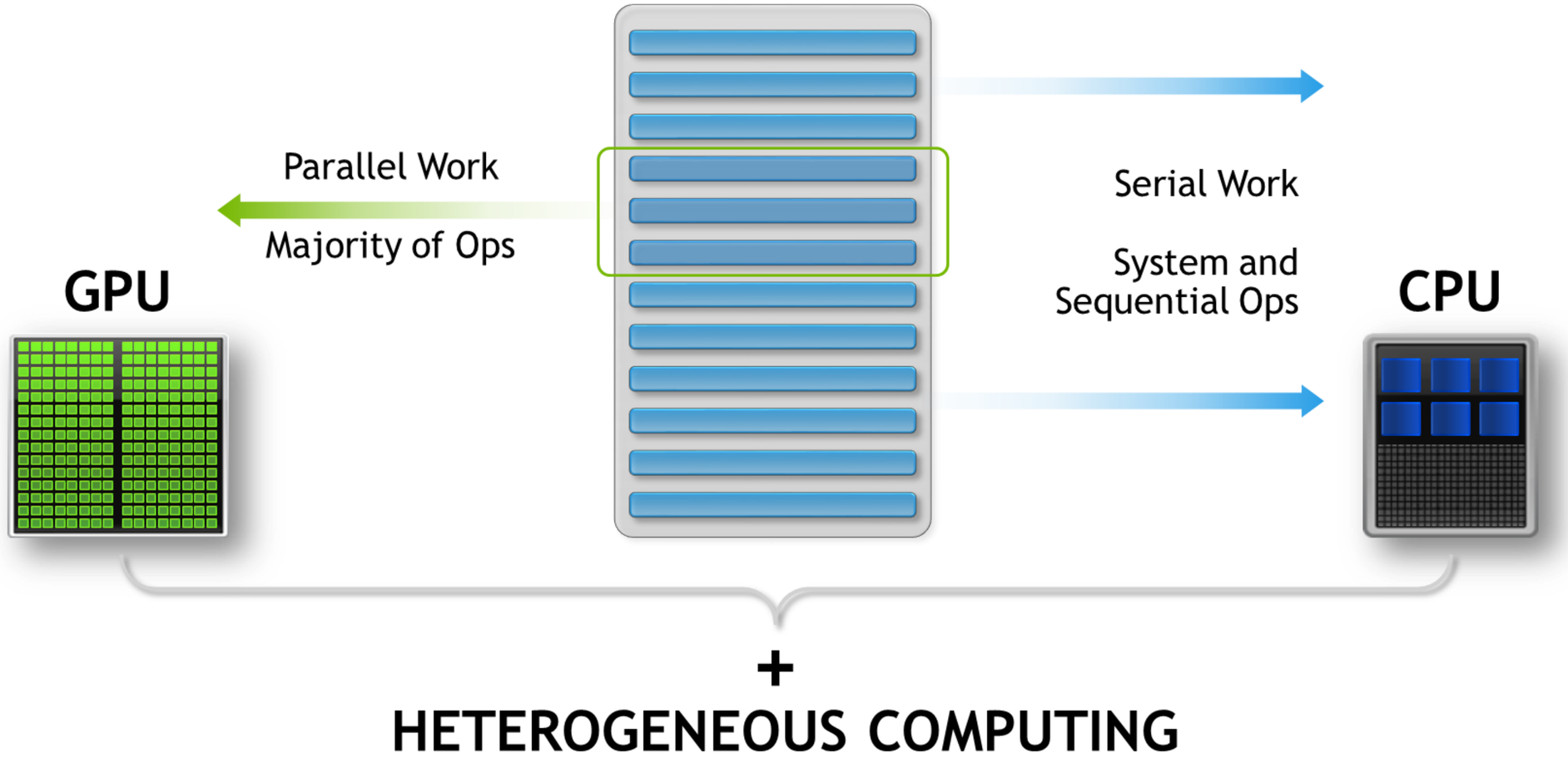


Accelerators are roadmap for Exa-scale computing

Almost all HPC clusters are equipped with GPUs

What changes as a programming HPC system evolves?

GPU revolutionises general purpose computing



Heterogeneity is key for modern hpc

AMPERE ALTRA MAX

128 cores/processor

Arm Neoverse N1

3.0 GHz

64 KB L1 cache

1 MB L2 cache

4 MB L3 cache

225 W

NVIDIA GRACE

144 cores/processor

Arm Neoverse V2

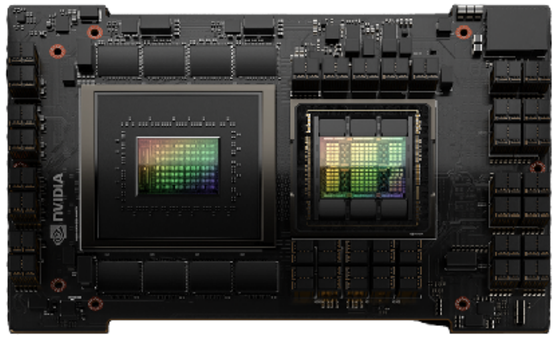
3.1 GHz

64 KB L1 cache

1 MB L2 cache

228 MB L3 cache

500 W



NVIDIA GRACE HOPPER

72 cores/processor

117 MB L3 cache

1 GPU H100

450 - 1000 W

SOPHON SG 2042

64 cores/processor

Risc-V

2.0 GHz

4 MB L1 cache

16 MB L2 cache

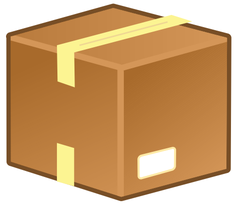
64 MB L3 cache

120 W

Summary: why GPU is trending?

Best theoretical FLOPs/$

Power efficient



Many FLOPs in one device
→ compact system possible

Why GPU has high throughput?

Definition: Throughput and Memory Bandwidth

Throughput

- Throughput in the context of GPU programming refers to the rate at which operations are completed or data is processed by the GPU

Memory Bandwidth

- Memory bandwidth refers to the rate at which data can be read from or written to the memory by the GPU
- Usually measured in gigabytes per second (GB/s)
- a critical factor in determining the performance of memory-intensive applications

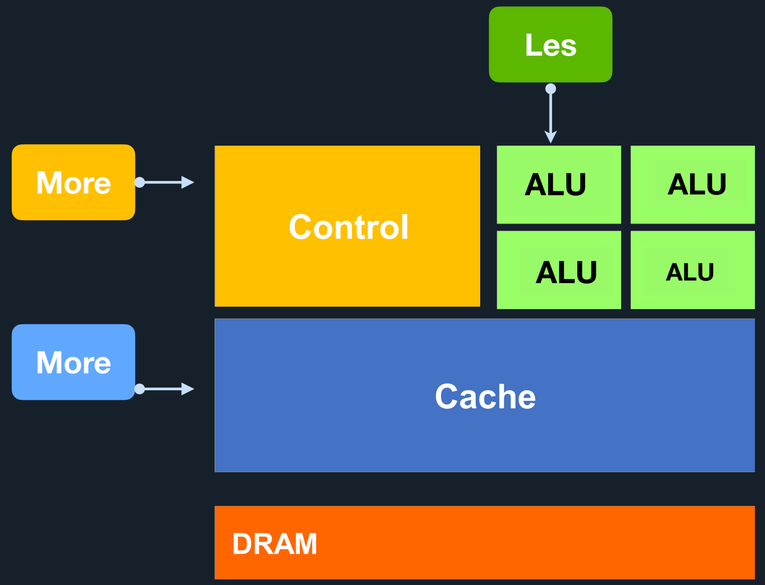
Theoretical Bandwidth: Calculated using hardware specifications such as memory clock rate and memory interface width. provides an upper bound on the memory performance of the device

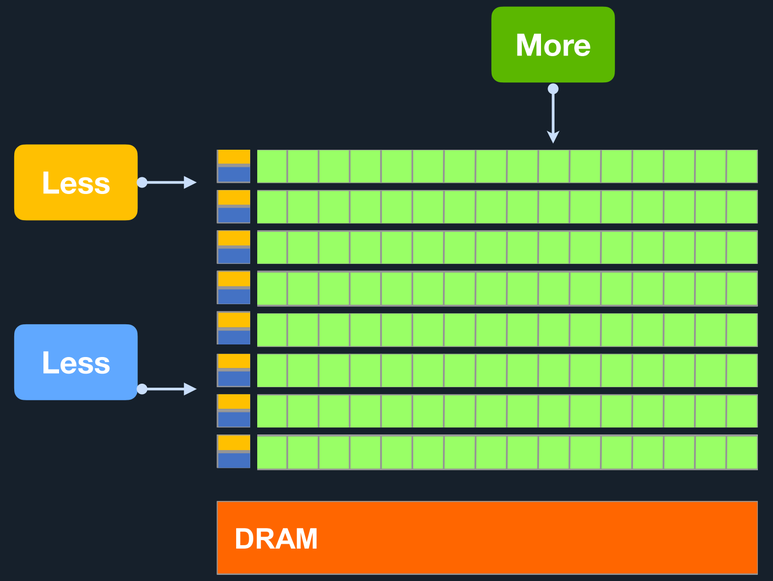
$$\text{Bandwidth} = \text{Memory Clock Rate} \times (\text{Memory Interface Width}/8) \times 2$$

Effective Bandwidth: Effective bandwidth is measured by timing specific program activities and understanding how data is accessed.

$$\text{Effective Bandwidth (GB/s)} = (\text{bytes read} + \text{bytes written}) \times 10^9 / \text{time elapsed}$$

CPU and GPU are designed to do specific task





Majority of silicon is dedicated to

- Several Several Arithmetic Logic Units (ALU)
- Large cache

Latency optimised via large caches

Majority of silicon is dedicated to

- Thousands of ALUs
- Each has its own control units and registers

High Throughput

Concurrency using CPU threads

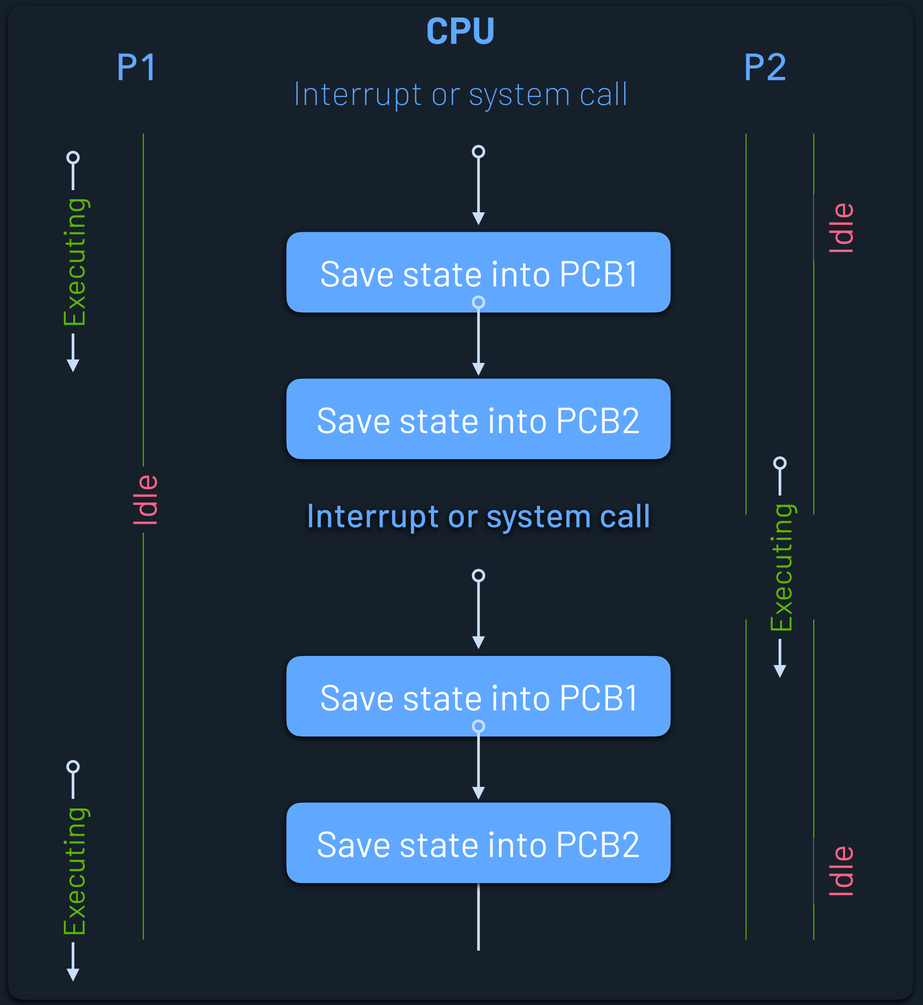
Modern CPUs often have multiple cores on chip

- each core has registers and ALU units to run a thread independently from other cores

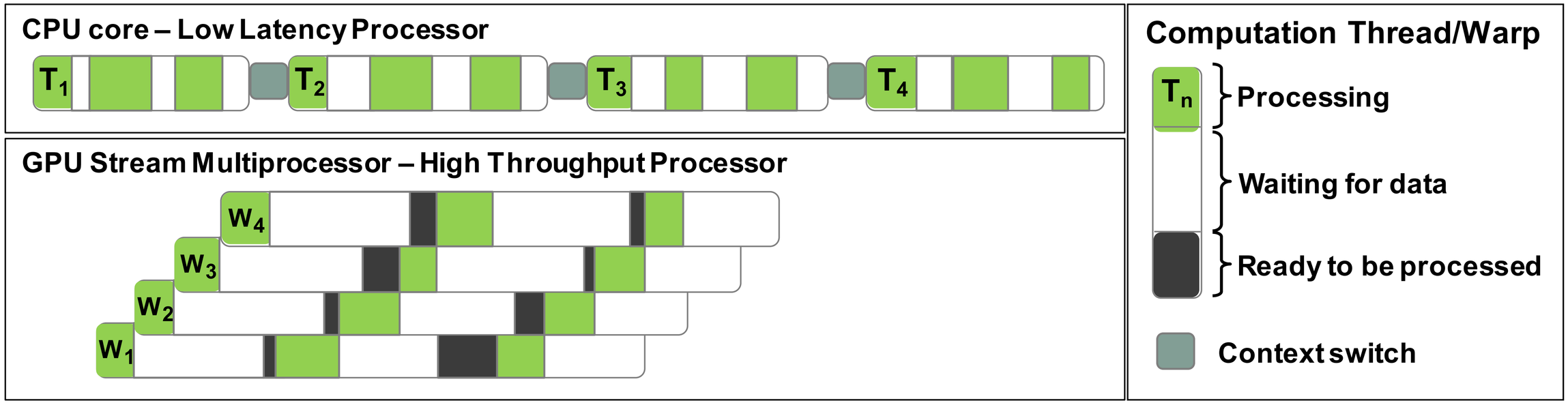
When there are more threads on the fly than available cores, the operating system can make a **context switch**

- the running thread is freezed: all information about its status is saved and put back for later restore
- a new thread take the hardware resources, load its previous status and restart its flow until a new context switch will take place

CPU threads context switch involves many backup operations, plus the fact that data is no longer in registers and caches



Low Latency or High Throughput



GPU hides memory latency

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

Speed vs Throughput

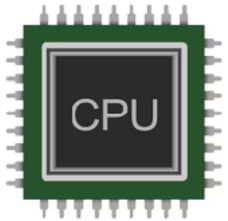
Speed

Throughput

Which is better depends on your needs…

Summary: Low Latency or High Throughput?

Optimal serial performance



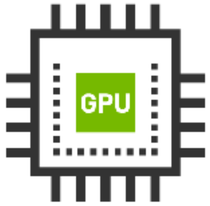
STRENGHTS

- Large main memory ~ 1TB+
- Faster clock speed ~ 4 GHz
- Latency optimised via large caches
- Small number of threads can run very quickly ~ Equal to number of cores

WEAKNESSES

- Low memory bandwidth ~200 GB/s
- Cache misses very costly
- Low performance/watt

Optimal parallel performance



STRENGHTS

- High bandwidth main memory ~ 1TB/s+
- Significant more compute resources ~ Few thousands cores
- High throughput
- High performance/watt

WEAKNESSES

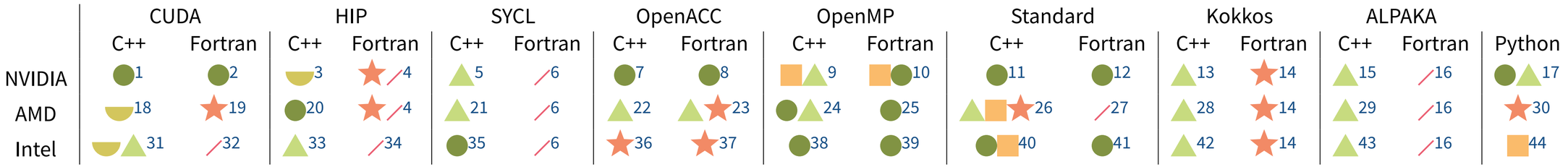
- Relatively low memory capacity ~80 GB per GPU (Total 640 GB unified in 8 GPU system)
- Low per-thread performance: 4 time slower clock speed than CPU core

Many and Models GPU Programming Model vs. Vendor Compatibility

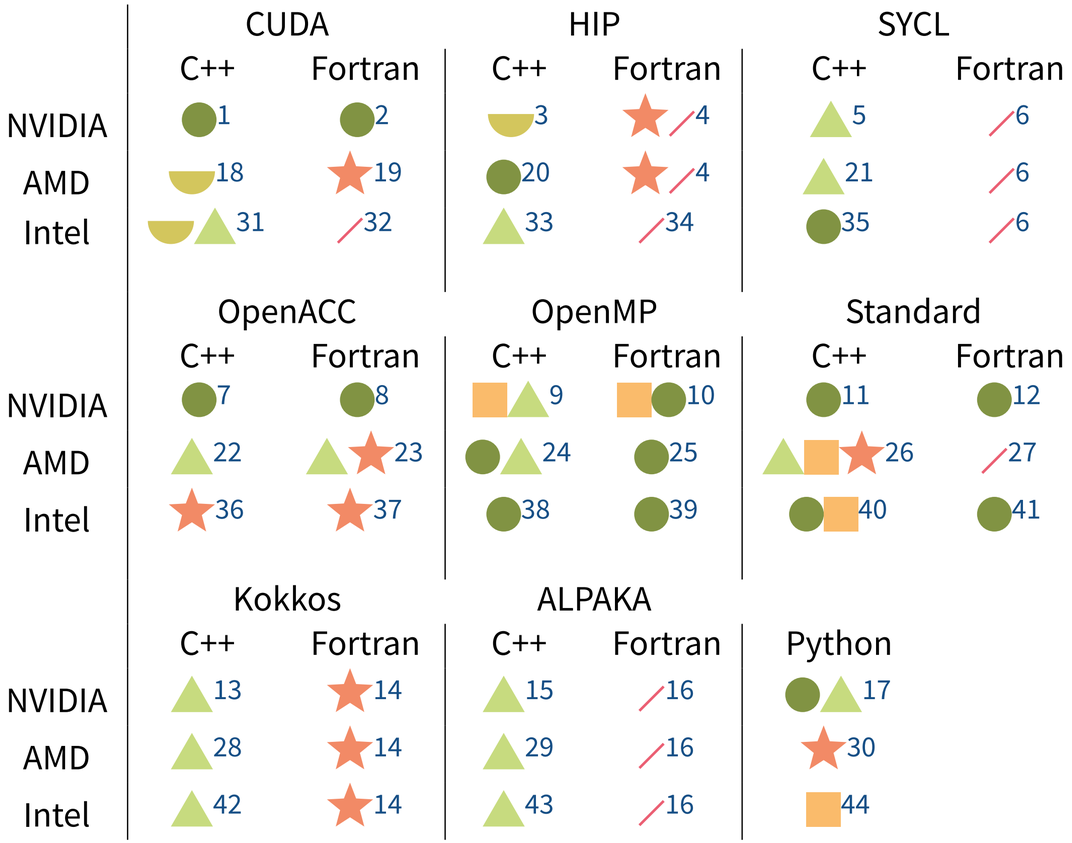
Thanks to @Andreas Hertena JSC

State of the GPUnion

- 3 vendors (AMD, Intel, NVIDIA), each with *native* programming model (HIP, SYCL, CUDA)
- Partly from community: OpenMP, OpenACC; Kokkos, RAJA, Alpaka
- Major languages: C/C++, Fortran
- Plethora of possibilities: 3 × 9 × 2 = 54 → What to choose?

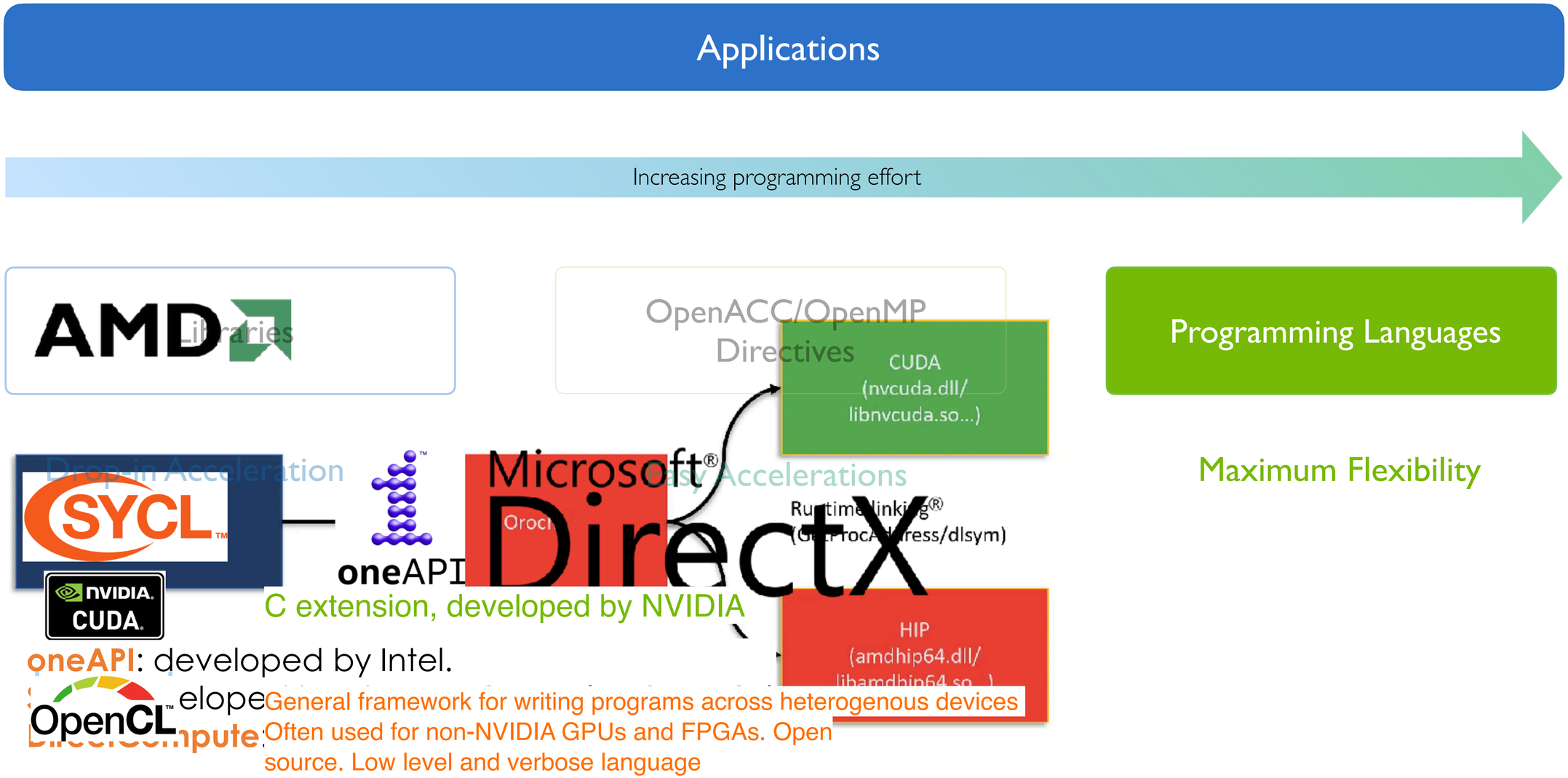


The table (split)

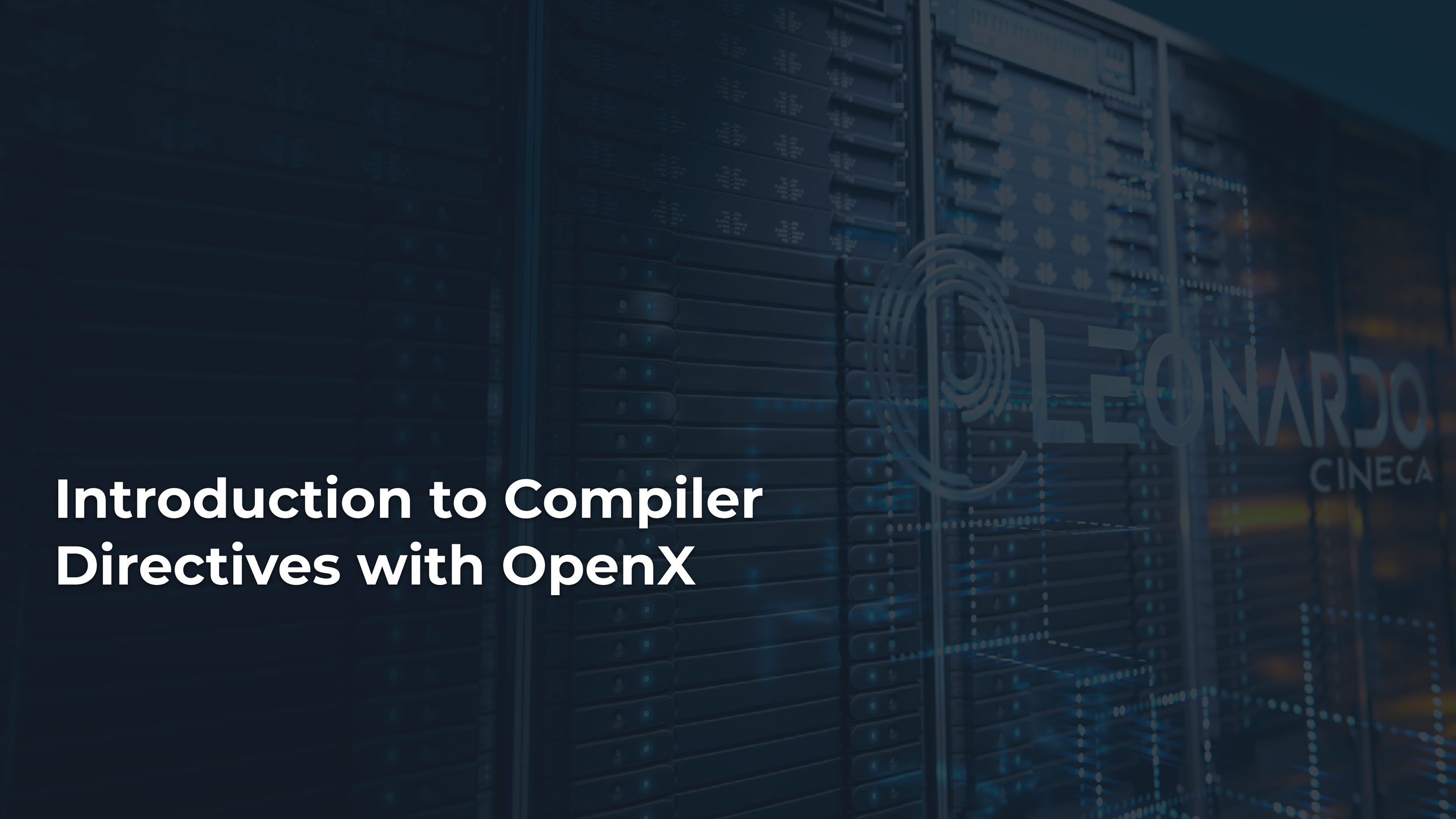


- State: Sep 2023 *evolving*
- Categories
  - Full vendor support
  - Indirect, but comprehensive support, by vendor
  - Vendor support, but not (yet) entirely comprehensive
  - Comprehensive support, but not by vendor
  - Limited, probably indirect support – but at least some
  - No direct support available
  - C++ C++ (sometimes also C)
  - Fortran Fortran

Three ways to accelerate application

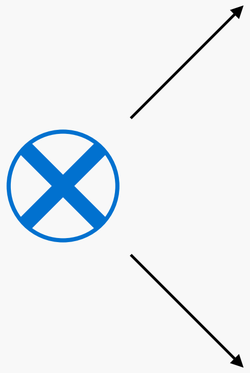


Introduction to Compiler Directives with OpenX



What is OpenX?

designed for performance and portability



OpenACC — More Science, Less Programming

Main focus is to target to offloading code onto GPUs

OpenMP

from v 4.0 allows offloading of tasks onto GPUs

IMPORTANT

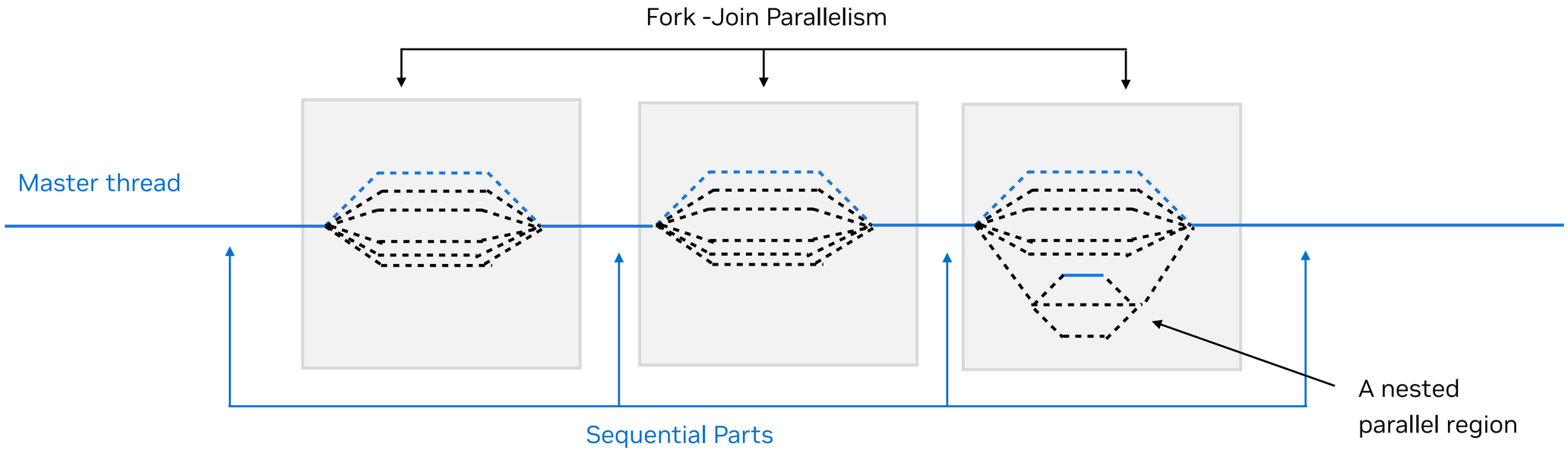
This course will not give you any concrete hints if Open**MP** is better than Open**ACC**

Are you familiar with directive based parallelism?

Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

- Directives are understood by OpenMP aware compilers (others are free to ignore)
- Generates parallel threaded code
  - Original thread becomes thread “0”
  - Share resources of the original thread (or rank)
  - Data-sharing attributes of variables can be specified based on usage patterns

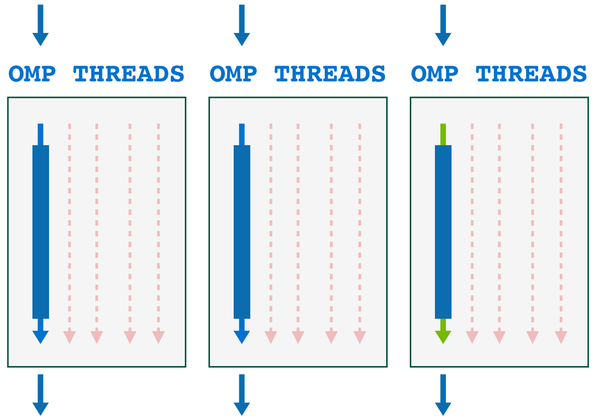


Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

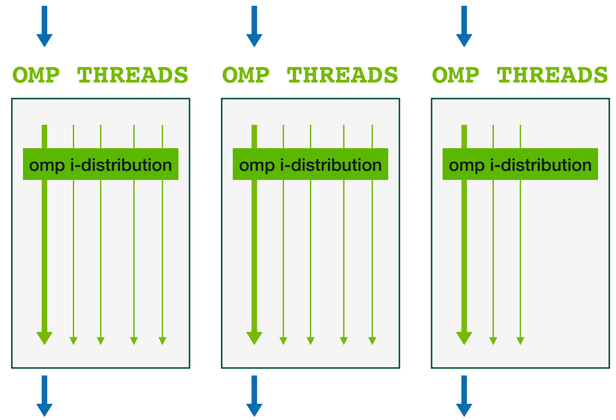
- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS

#pragma omp parallel



All threads will execute the region

#pragma omp parallel for



All threads will execute a part of the iterations

Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

Serial

```
for (int i = 0; i < N; ++i)
{
   C[i] = A[i] + B[i];
}
```

- 1 thread/process will execute each iteration sequentially
- Total time = time_for_single_iteration * N

Parallel

```
#pragma omp parallel
for (int i = 0; i < N; ++i)
{
   C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration redundantly (overwriting values of C)
- Total time = time_for_single_iteration * N

Parallel worksharing

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
   C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration (roughly N/4 per thread)
- Total time = time_for_single_iteration * N/4

What do you need to start with?

Compilers	Fortran	C/C++
GNU	gfortran *.f90 *-fopenmp*	gcc/g++ *.c/c++ *-fopenmp*
Intel	ifort *.f90 *-qopenmp*	icc/icpc *.c *-qopenmp*
PGI	pgf90 *.f90 *-mp*	pgcc *.c *-mp*

GNU Fortran compiler OpenMP command line switch

No need to include the library if only using the compiler directives. The library only gets you the API calls.

NVIDIA’s HPC Compilers (AKS PGI)

flag: -mp

- `nvc -mp -fast -Minfo=mp my_program.c`
- `nvc++ -mp -fast -Minfo=mp my_program.cpp`
- `nvfortran -mp -fast -Minfo=mp  my_program.cf90`

Task-1: Parallelise a serial Laplace 2D

Code description

```
while (error > tol && niter < niter_max)
error = 0.0;

 for (int j = 1; j < n-1; ++j) {
  for (int i = 1; i < m-1; ++i) {
   Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m]
   + A[idx+m]);

   error = fmax(error, fabs(Anew[idx] - A[idx])); }
   }
 for (int j = 1; j < n-1; ++j)
  for (int i = 1; i < m-1; ++i)
   A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

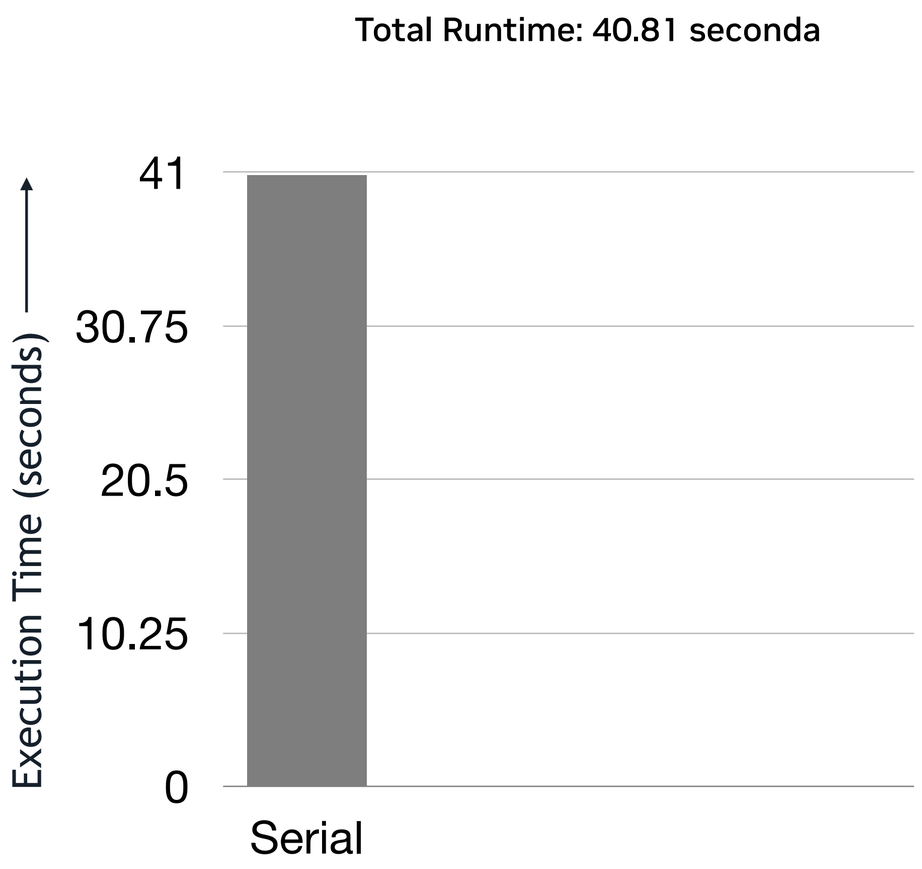
Compute max error for converge

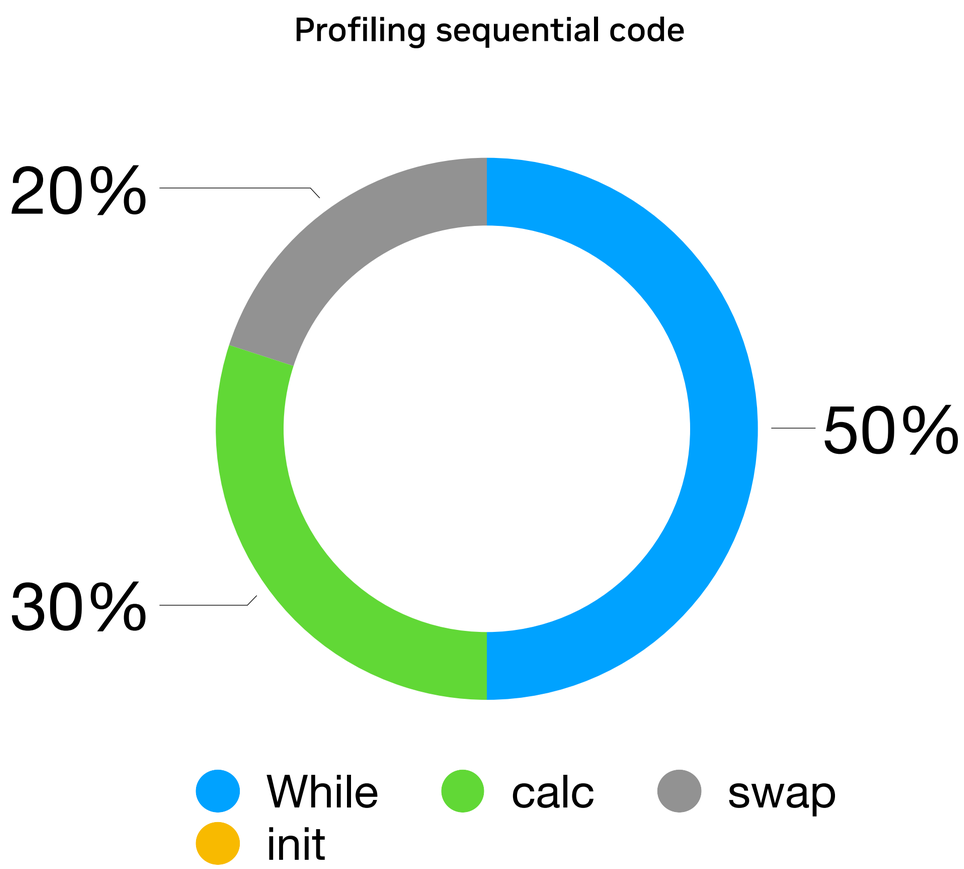
Compute max error for converge

Swap input/output arrays

Hotspots: Identify the portions of code that took the longest to run

Simulation was performed 1000 Iterations





00-laplace2d-openmp: parallelize with OpenMP

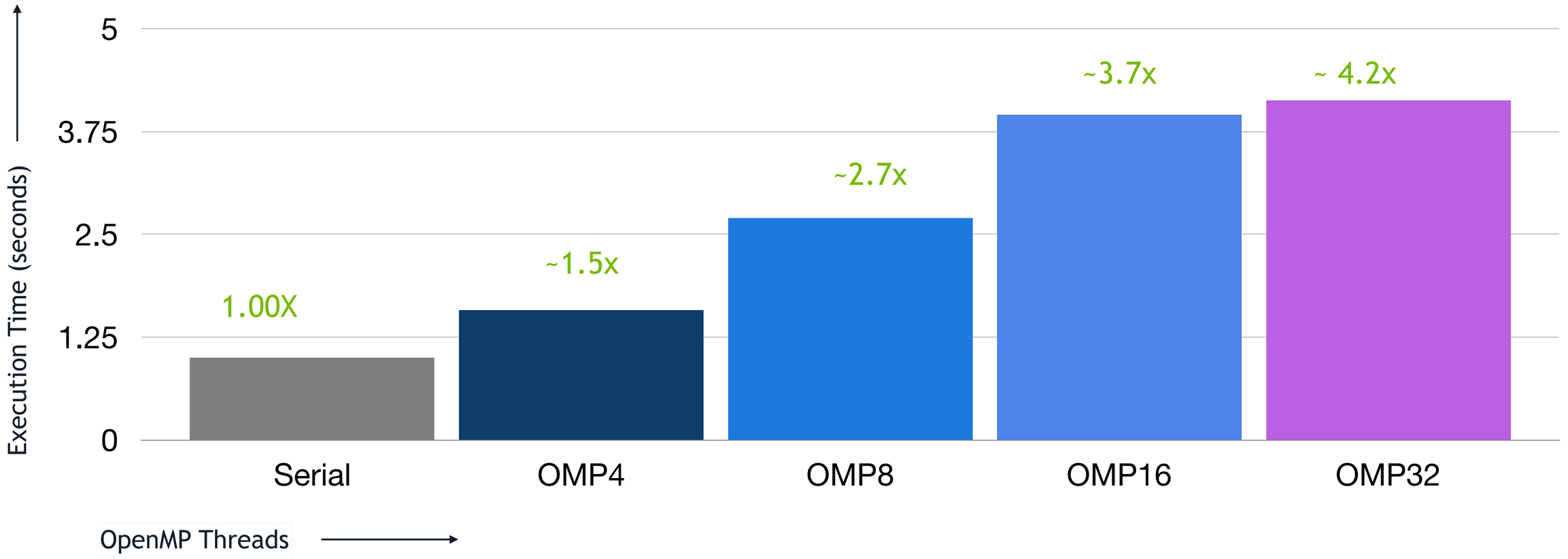
```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

00-laplace2d-openmp: parallelize with OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for collapse(2) shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for collapse(2) shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

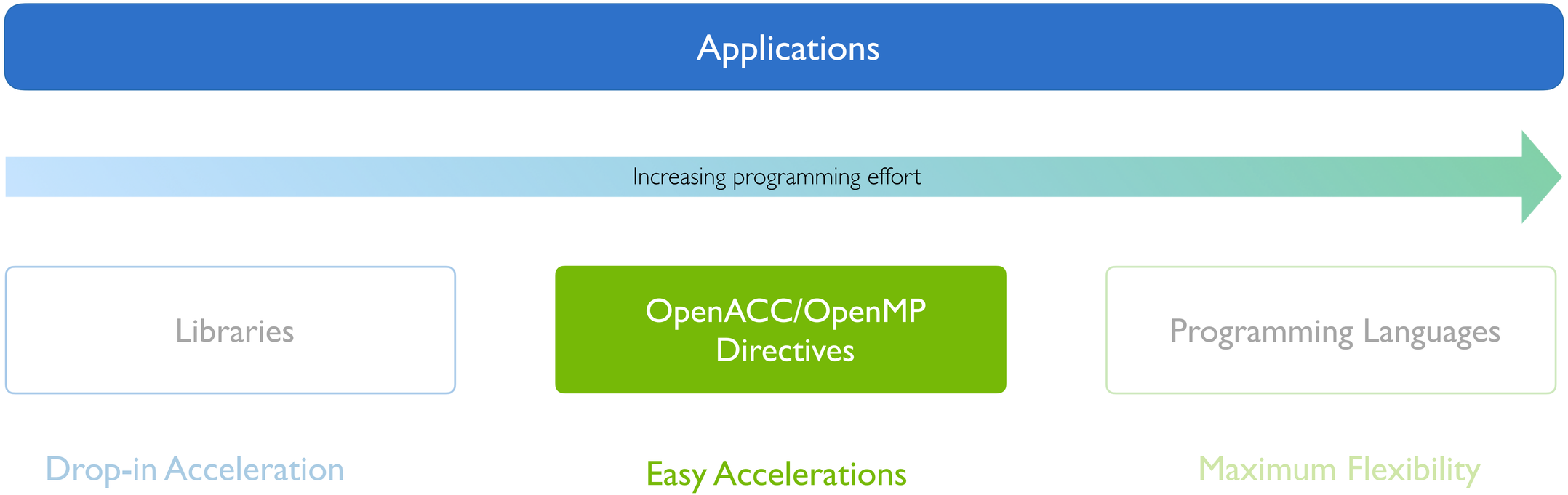
Performance speed up (higher is better)

Simulation was performed 1000 Iterations



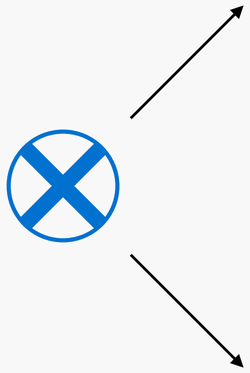
Will your legacy OpenMP code perform well on the GPU?

Three ways to accelerate application



What is OpenX?

designed for performance and portability



OpenACC — More Science, Less Programming

Main focus is to target to offloading code onto GPUs

OpenMP

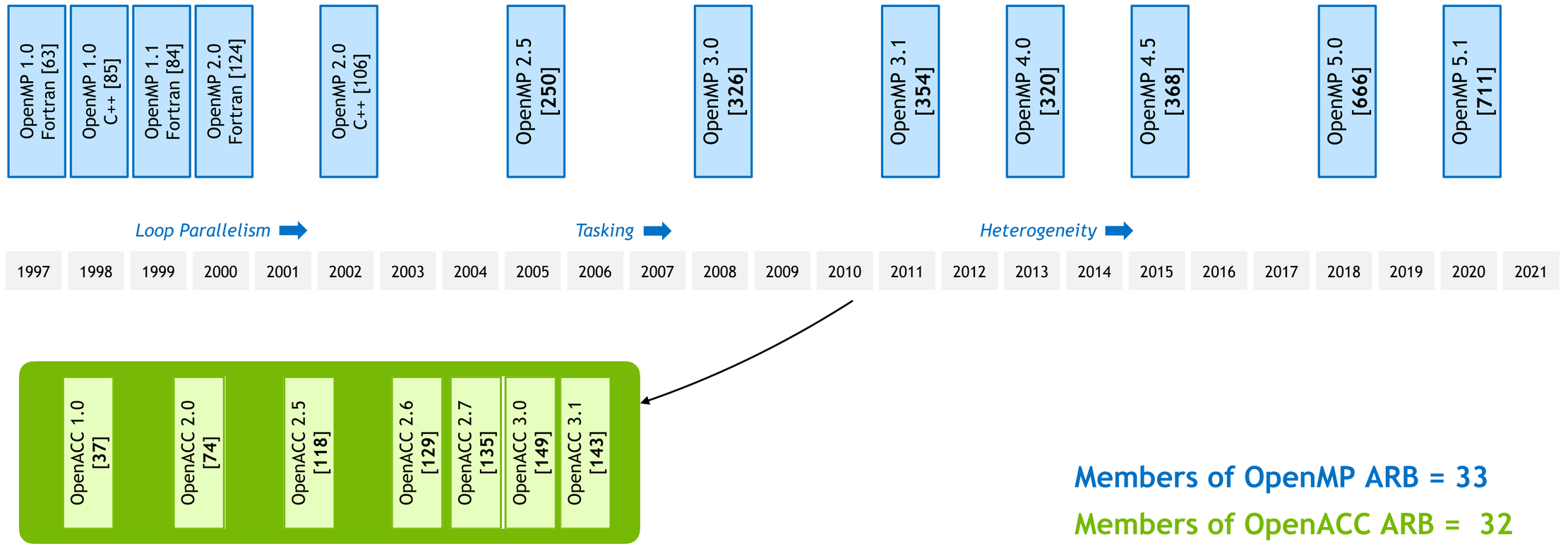
from v 4.0 allows offloading of tasks onto GPUs

IMPORTANT

This course will not give you any concrete hints if Open**MP** is better than Open**ACC**

OpenX (X = OMP, ACC): 1997-2021

Looking at TIME (pace of innovation) and SPACE (specification length)



Members of OpenMP ARB = 33

Members of OpenACC ARB = 32

OpenX (X = OMP, ACC): 1997-2021

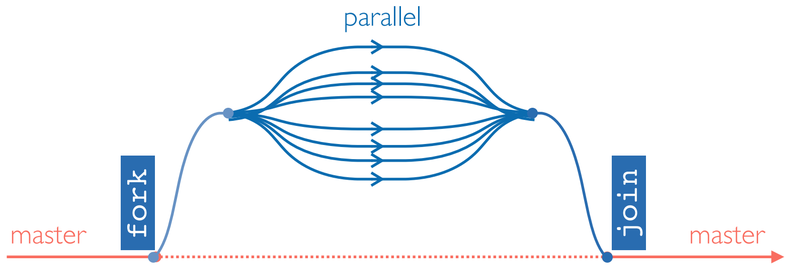
Same basic principle: Fork-Join model

OpenACC

Specifically targets GPU accelerators

It started after OpenMP

Basic principle: fork-join model



OpenACC is more descriptive

Compiler support

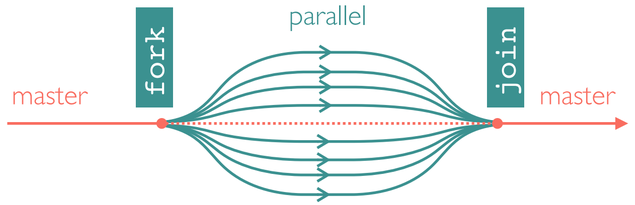
PGI, Cray, NVIDIA

OpenMP

Designed to replace low-level and multi-threaded programming solutions like POSIX, threads or Pthreads

Intend to target independent processor (shared memory)

Basic principle: fork-join model



OpenMP 4./4.5 onwards; offloading capabilities

OpenMP is more prescriptive

Compiler support

GCC, Intel, IBM XL, LLVM/Clang etc

Porting/Offloadings Application with OpenX

Instructions to the compiler on how to compile the code

OpenACC

```
/* C/C++ code to offload on the device*/
#pragma acc directive-name [clause-list]
  structured-block
```

```
! Fortran code to offload on the device
 !$acc directive-name [clause-list]
   structured-block
 !$acc end directive-name
```

OpenMP

```
/* C/C++ code to offload on the device*/
#pragma mp directive-name [clause-list]
  structured-block
```

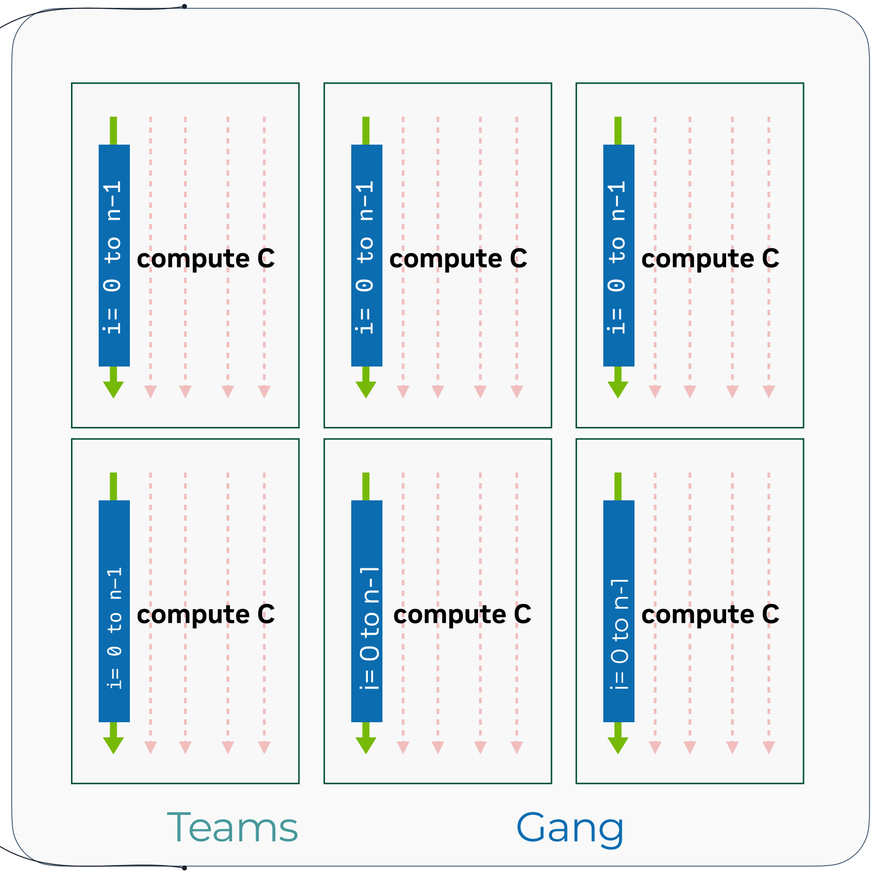
```
! Fortran code to offload on the device
 !$mp directive-name [clause-list]
   structured-block
 !$mp end directive-name
```

Expressing parallelism with OpenX parallel directives

Execute a compute region on a device

```
#pragma acc parallel
#pragma omp target teams
{
  Compiler will generate 1 or more parallel GANGS or Teams of
  threads which execute block of code redundantly
}
```

```
!$acc parallel
!$omp target teams
DO I = 1, N
  Compiler will generate 1 or more parallel GANGS or Teams of threads
  which execute block of code redundantly
END DO
!$omp end target teams
!$acc end parallel
```



Introduction to OpenACC

Objectives

What are Compiler Directives?

Easy way porting code to GPU

What is OpenACC?

Execution Model

Data Locality

Memory model

Case study

What is OpenACC?

OpenACC is …

a directive-based

parallel programming model

designed for

performance and portability

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

Incremental

Almost all the compilers supports OpenX

```
<sequential part>

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }
```

Sequential code

Parallel region

Parallel region

Verify results

- Maintain existing sequential code
- Add annotations to expose parallelism
- Verifying correctness, annotate mode of the code

Single source

Almost all the compilers supports OpenX

```
<sequential part>

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }
```

Sequential code

Parallel region

Parallel region

Verify results

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine while sequential code is maintained

Low learning curve

Compiler hints

```
<sequential part>

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }

#pragma X parallel clause
  for( i = 0; i < N; i++ )
  {
      < loop code >
  }
```

Sequential code

Parallel region

Parallel region

Verify results

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine while sequential code is maintained

Why do you choose OpenX?

Enhancing Code Productivity and Portability

Pros of OpenX

- Easy to use
- Portability
- Productivity
- Incremental Parallelisation

Cons of OpenX

- Performance Overheads
- Limited Control
- Limited Features Compared to CUDA
- Vendor and Hardware Support

GPU programming can be thought of as a two-step process

Compute Bound

- Exposing to the GPU environment

Data Bound

- Data management directives

Expressing Parallelism with OpenACC

From now we focus only on OpenACC

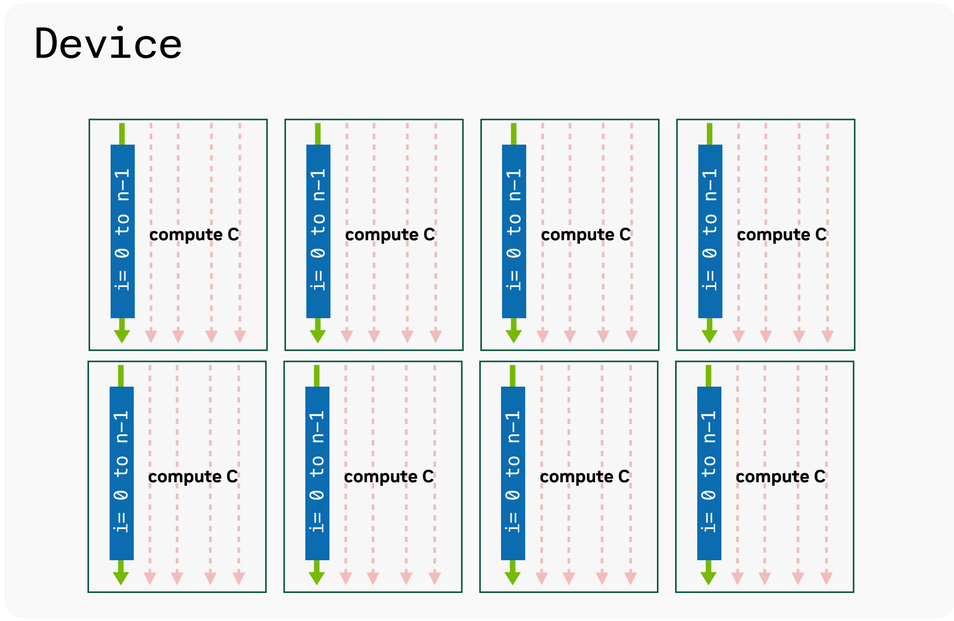
Parallelism identified by programer

Parallel Loop Directives

Parallel: a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

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

```
#pragma acc parallel
  for (int i=0; i<N; i++)
   c[i] = a[i] + b[i]
```



Parallelism identified by programer

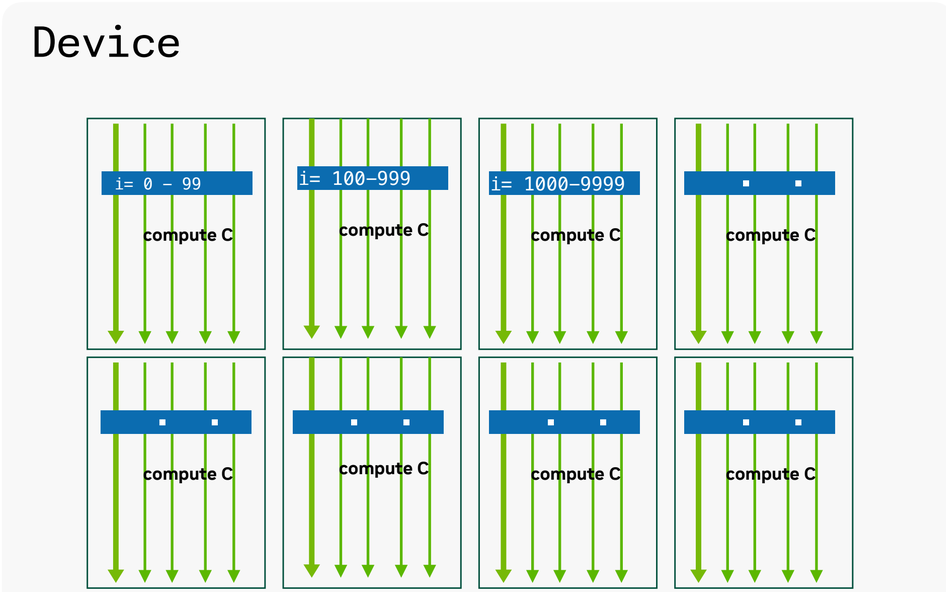
Parallel Loop Directives

Parallel: a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

Loop: identifies a loop that should be distributed across threads. Parallel and loop are often placed together

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

```
#pragma acc parallel
{
  for (int i=0; i<N; i++)
   a[i] = 0;

  #pragma acc loop
  for (int i=0; i<N; i++)
   a[i]++;
}
```



Parallelism identified by programer

Parallel Loop Directives

Parallel: a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

Loop: identifies a loop that should be distributed across threads. Parallel and loop are often placed together

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

```
#pragma acc parallel loop
  for (int i=0; i<N; i++)
   c[i] = a[i] +b[i]
```

Similar to OpenMP, compiler translates the parallel region into a kernel that runs in parallel on the GPU

OpenACC Loop construct

Parallelising loop nests

Nest loop

- Multiple loop directives to parallelise multi-dimensional loop nests
- This will allow you to express more levels of parallelism and increase performance further
- In this case, inner loop directives may be ignored

```
#pragma acc parallel loop
{
  #pragma acc loop
  for (int i=0; i<N; i++)
   #pragma acc loop
    for (int j=0; j<M; j++)
          a[i][j] = 0;
}
```

OpenACC kernel construct

Compiler determines what can be safely parallelised

With the kernels directive, the loop directive is implied

The programmer can still explicitly define loops with the loop directive, however this could affect the optimizations the compiler makes

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

```
#pragma acc kernel
{
  #pragma acc loop
  for (int i=0; i<N; i++)
   a[i] = 0;
  #pragma acc loop
  for (int i=0; i<N; i++)
   a[i]++;
}
```

No explicit counterpart

Kernels Vs Parallel Construct

When fully optimised both will give similar performance

Kernel compute directives

- Compiler’s leeway parallelising and optimising a loop
- Only provide by OpenACC
- Compiler guarantees correctness
- Or at least it does two things:
  - Fuse the two loop nests into one loop nest
  - Generates two kernels
- Implicit barrier at the end and between each loop

Parallel compute directives

- Programmer’s responsibility to identify region for parallelization
- Programmer guarantees correctness
- Parallel just run the same code on multiple threads redundantly
- Guarantees that no dependency occurs iterations
- Implicit barrier at the end of the parallel region

Compiling OpenACC code with NVHPC

Building and running enabled OpenACC code

The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting OpenACC C and Fortran

- nvc -fast -gpu=target architecture -Minfo=accel -o laplace_2d laplace_2d.c

Compiling OpenACC program using GCC

- gcc -fast -fopenacc -foffload=offload target -o laplace_2d laplace_2d.c

Compiling OpenACC program using CRAY

- gcc -fast -f pragma=acc -h msgs -o laplace_2d laplace_2d.c

Building and running enabled OpenACC code

Flags	Specification
-acc	enable OpenMP targeting device
-acc=host	to generate an executable that will run serially on the host CPU
-acc=multicore	parallelize for a multicore CPU
-acc=gpu –gpu=cc80	map OpenACC parallelism to an NVIDIA GPU, compile targeting compute capability
–gpu=managed	place all allocatables in CUDA Unified Memory
–gpu=pinned	use CUDA pinned memory for all allocatables
-Minfo=acc?all	Compiler diagnostics for OpenACC
export NVCOMPILER_ACC_NOTIFY = 1|2|3	Environment variable for NOTIFY

```
$ nccx -acc -gpu=cc80,managed -Minfo=acc -o binary OpenACC_Code.c
```

Building and running enabled OpenACC code

Flags	Specification
-mp	enable OpenMP targeting device
-mp=gpu	to generate an executable that will run serially on the host CPU
–gpu=cc80	map OpenACC parallelism to an NVIDIA GPU, compile targeting compute capability
–gpu=managed	place all allocatables in CUDA Unified Memory
–gpu=pinned	use CUDA pinned memory for all allocatables
-Minfo=acc?all	Compiler diagnostics for OpenACC
export NVCOMPILER_ACC_NOTIFY = 1|2|3	Environment variable for NOTIFY

```
$ nccx -mp=gpu -gpu=cc80,managed -Minfo=mp -o binary OpenMP_Code.c
```

Task-2: Add Parallel directives

01-laplace2d_OpenACC_parallel: parallelize with OpenACC

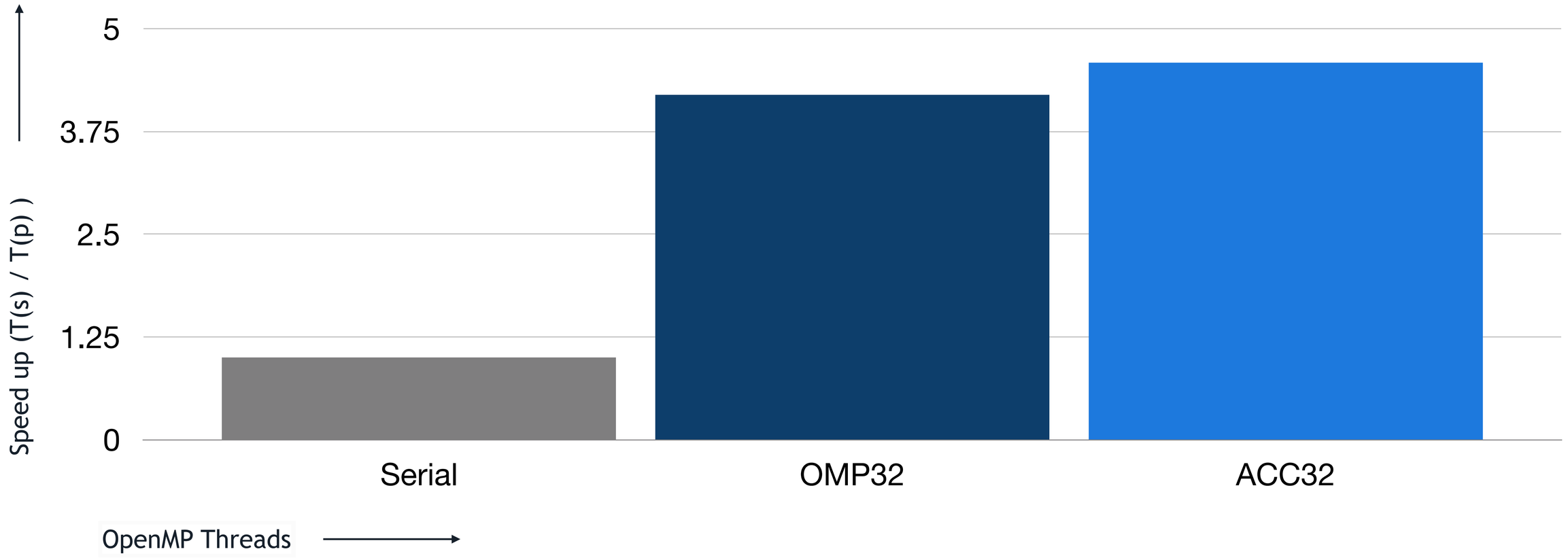
```
while ( error > tol && iter < iter_max ) {
  error=0.0;

for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

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

Things to do

- Make these two loops parallel with OpenACC directives
- Add Parallel and Kernel construct
- Hint you might want to use reduction as well

Performance speed up (higher is better)

Simulation was performed 1000 Iterations on Leonardo



01-laplace2d_OpenACC_parallel: parallelize with OpenACC

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Laplace2D_kernels_report

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Implicit reduction

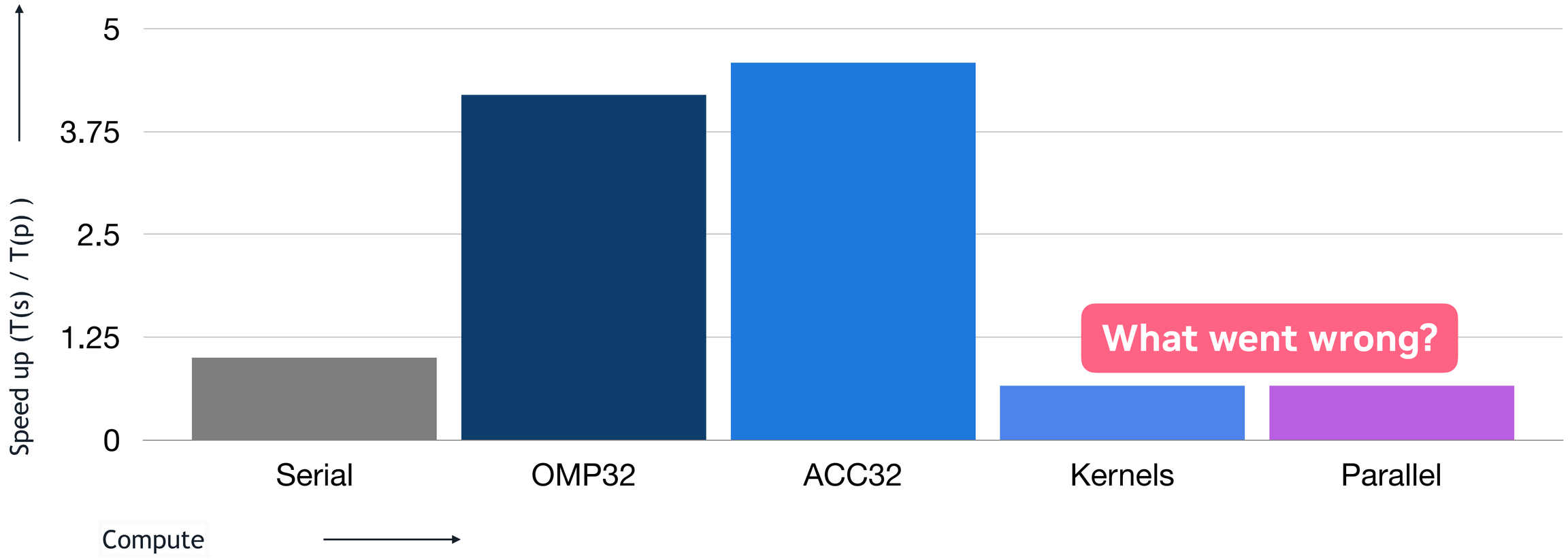
Laplace2D_kernels_report

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Compiler Generates reduction

Performance speed up (higher is better)

Simulation was performed 1000 Iterations on Leonardo



Profile-Driven Development

OpenX (X = OMP, ACC) porting strategies

Allows programmers to develop threaded parallel codes on shared memory computational units

Identify the compute kernels

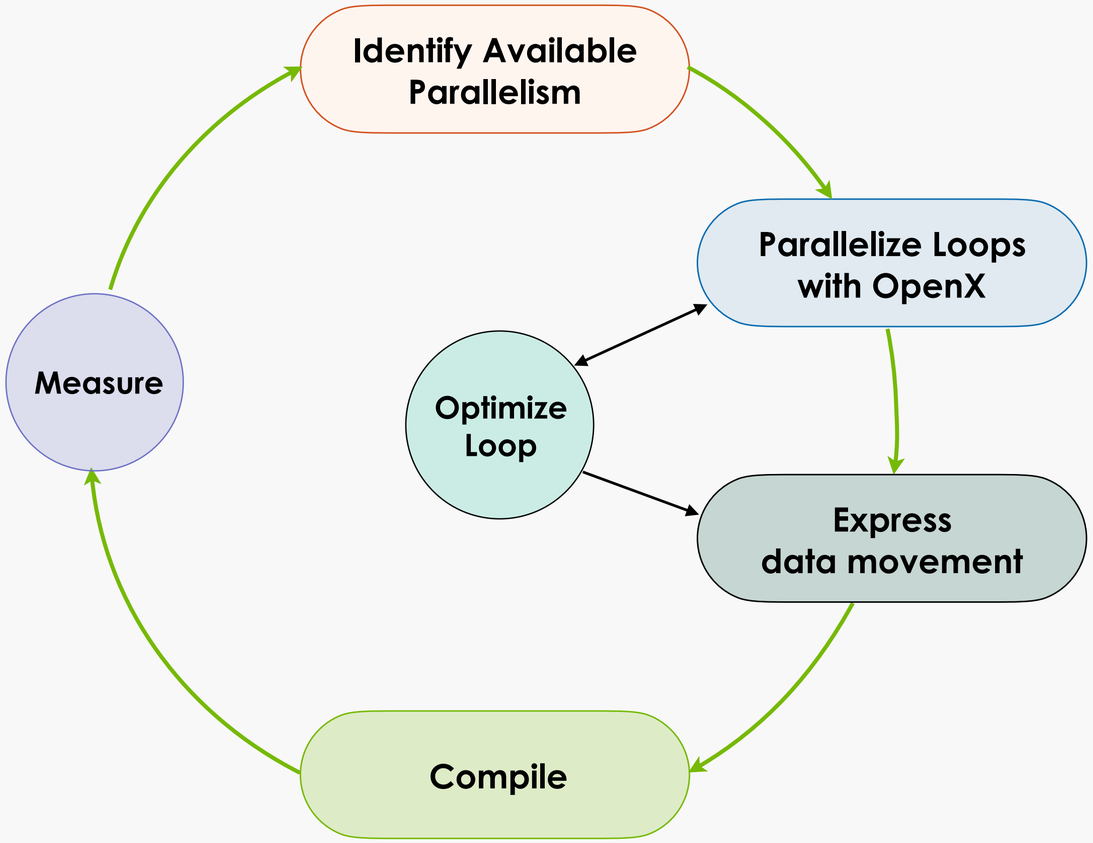
- most compute intensive code
- use performance analysis tools to find bottlenecks

Expressing parallelism within the kernel

- Use performance analysis tools to find bottlenecks
- Track independent work units with well define data access

Managing data transfer between CPU and Device

- relevant data needs to be moved from the host memory to device memory
- kernel executes using device memory
- relevant data needs to be moved from device to host memory



Tools we will use: NSIGHT SUITE

Workflow

Nsight product family

Nsight System

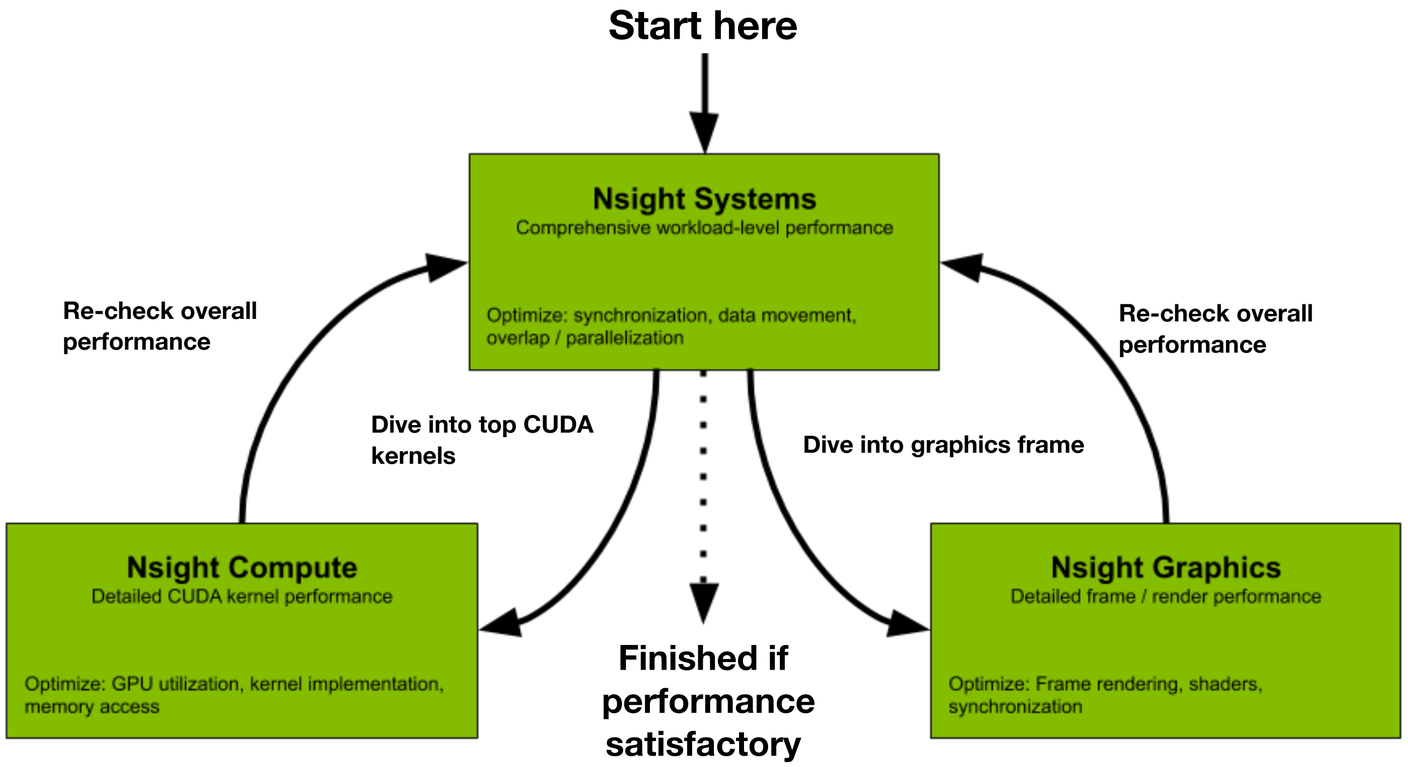
Analyze application algorithm system-wide

Nsight Compute

Debug/ Optimise CUDA kernel

Nsight Graphics

Debug/ Optimise graphics workloads



Courtesy: Mozhgan Kabiri chimeh, NVIDIA

Tools we will use: NSIGHT SUITE

Workflow

Application-wide profiling (Systems), Kernel-level profiling (Compute)

Instrument with NVIDIA Tools Extension (NVTX):

Automatic or manual

- Create (nested) ranges, define macros
- Compiler instrumentation

Tracing: CUDA API calls, NVTX trace

Sampling, hardware counters

NVTX primer: https://devblogs.nvidia.com/parallelforall/cuda-pro- tip-generate-custom-application-profile-timelines-nvtx/

Nsight Systems

Nsight Compute

Courtesy: Mozhgan Kabiri chimeh, NVIDIA

NSIGHT: Recording an application timeline

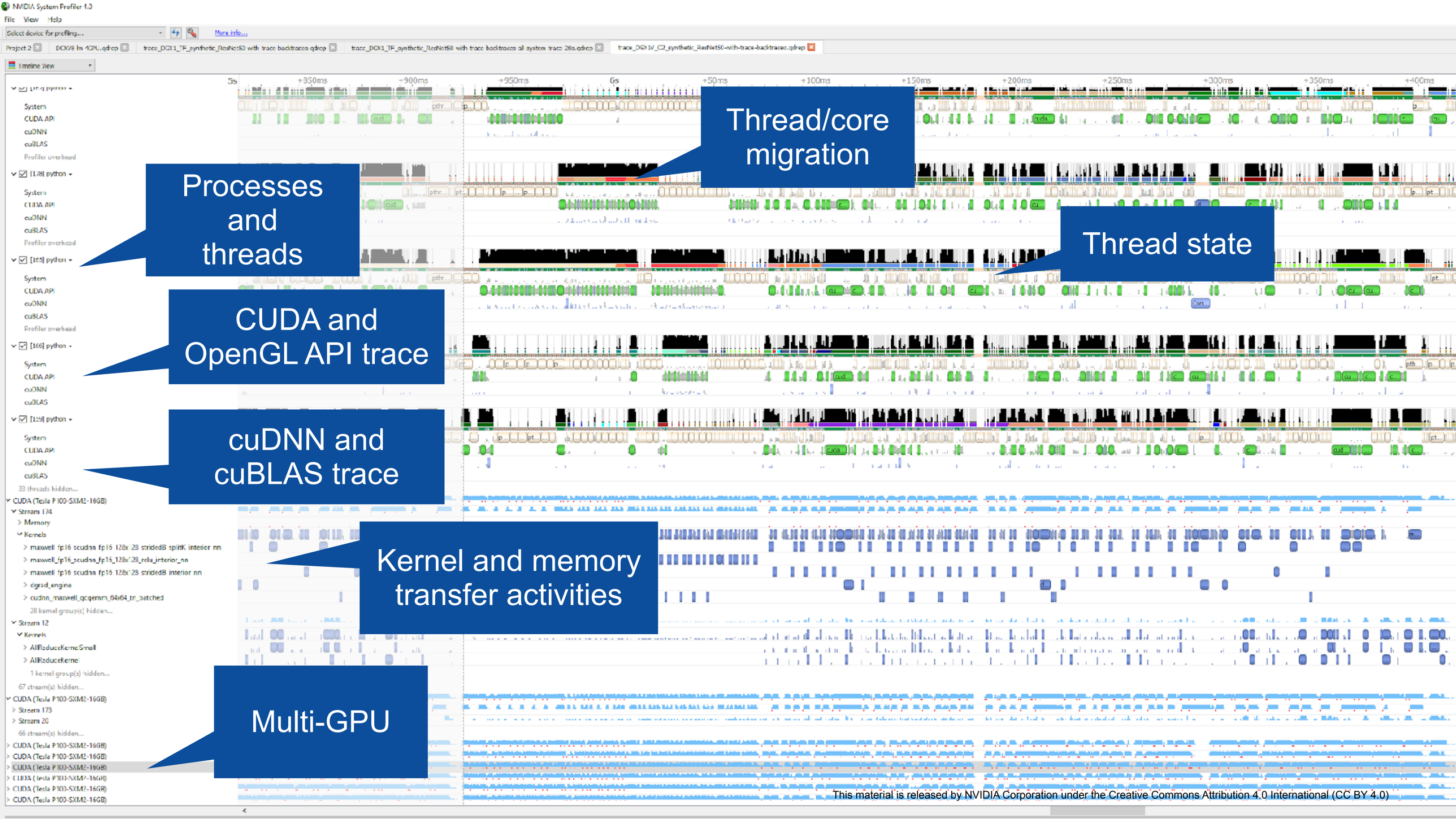
Notable flags for nays profile

nsys profile -t **cuda,nvtx,mpi,openmp** --stats=true --force-overwrite true -o my_report ./myapp

- profile – start a profiling session
- -t: Selects the APIs to be traced (cuda, cublas, nvtx, mpi openmp and openacc in this example)
- —cuda-memory-usage = true or false
- --stats: if true, it generates summary of statistics after the collection
- --force-overwrite: if true, it overwrites the existing generated report
- -o – name for the intermediate result file, created at the end of the collection (.qdrep filename)

nsys --help or nsys [specific command] --help

Inspect results: Open the report file in the GUI
See also https://docs.nvidia.com/nsight-systems/



Hotspots: Identify the portions of code that took the longest to run

Simulation was performed 1000 Iterations

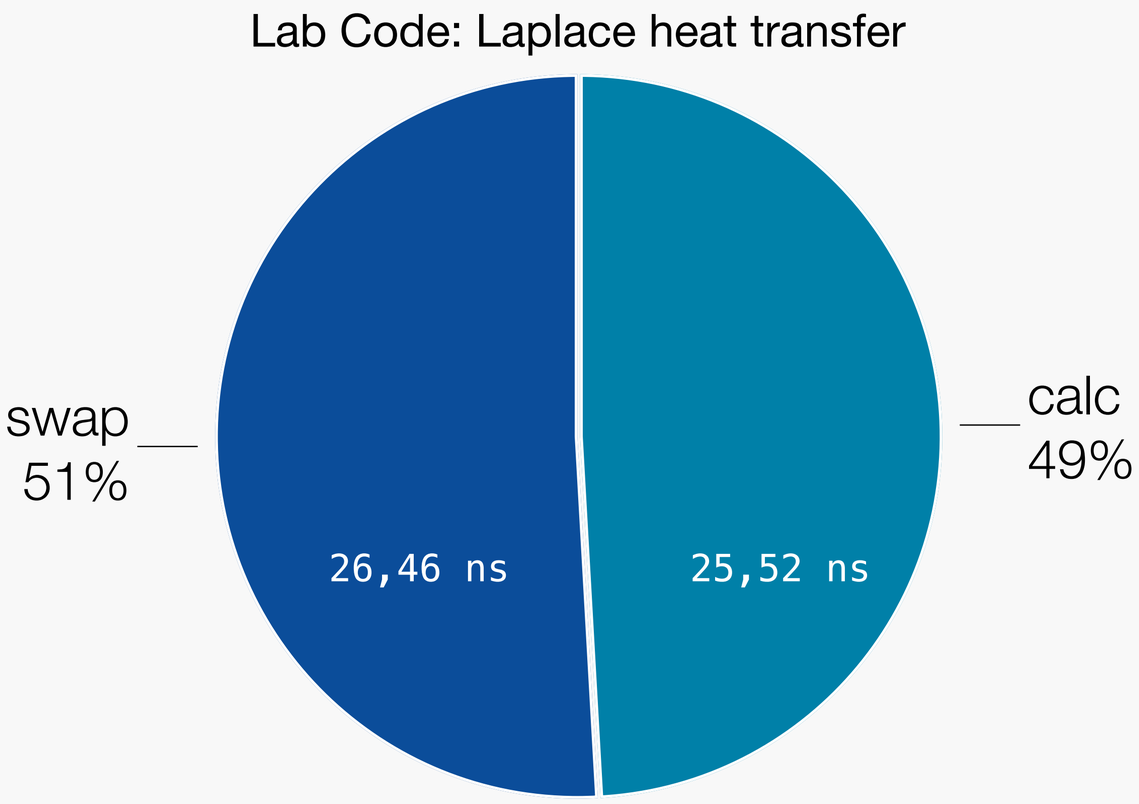
Profiling code

- Obtained information about how the code ran

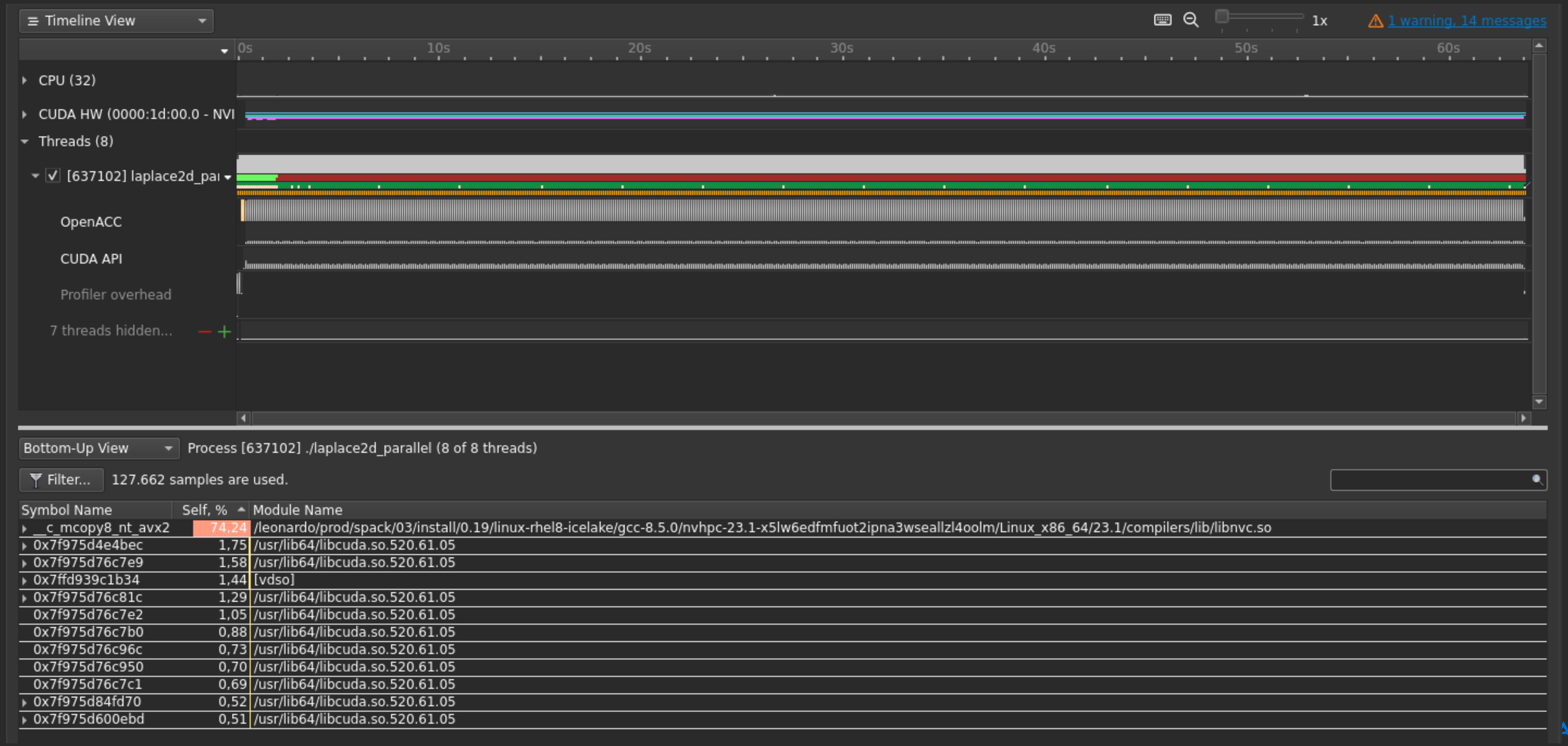
This includes

- Total runtime
- Runtime of individual routines
- Hardware counters
- Use Profiling tools

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing



NVIDIA Sight System



Symbol Name	Self, %	Module Name
__c_mcopy8_nt_avx2	74,24	/leonardo/prod/spack/03/install/0.19/linux-rhel8-icelake/gcc-8.5.0/nvhpc-23.1-x5lw6edfmfuot2ipna3wseallzl4oolm/Linux_x86_64/23.1/compilers/lib/libnvc.so
0x7f975d4e4bec	1,75	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c7e9	1,58	/usr/lib64/libcuda.so.520.61.05
0x7ffd939c1b34	1,44	[vdso]
0x7f975d76c81c	1,29	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c7e2	1,05	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c7b0	0,88	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c96c	0,73	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c950	0,70	/usr/lib64/libcuda.so.520.61.05
0x7f975d76c7c1	0,69	/usr/lib64/libcuda.so.520.61.05
0x7f975d84fd70	0,52	/usr/lib64/libcuda.so.520.61.05
0x7f975d600ebd	0,51	/usr/lib64/libcuda.so.520.61.05

Data Management with OpenACC

From now we focus only on OpenACC

Contents

- CPU vs GPU Memories
- CUDA Unified (Managed) Memory
- OpenACC Data Mangement

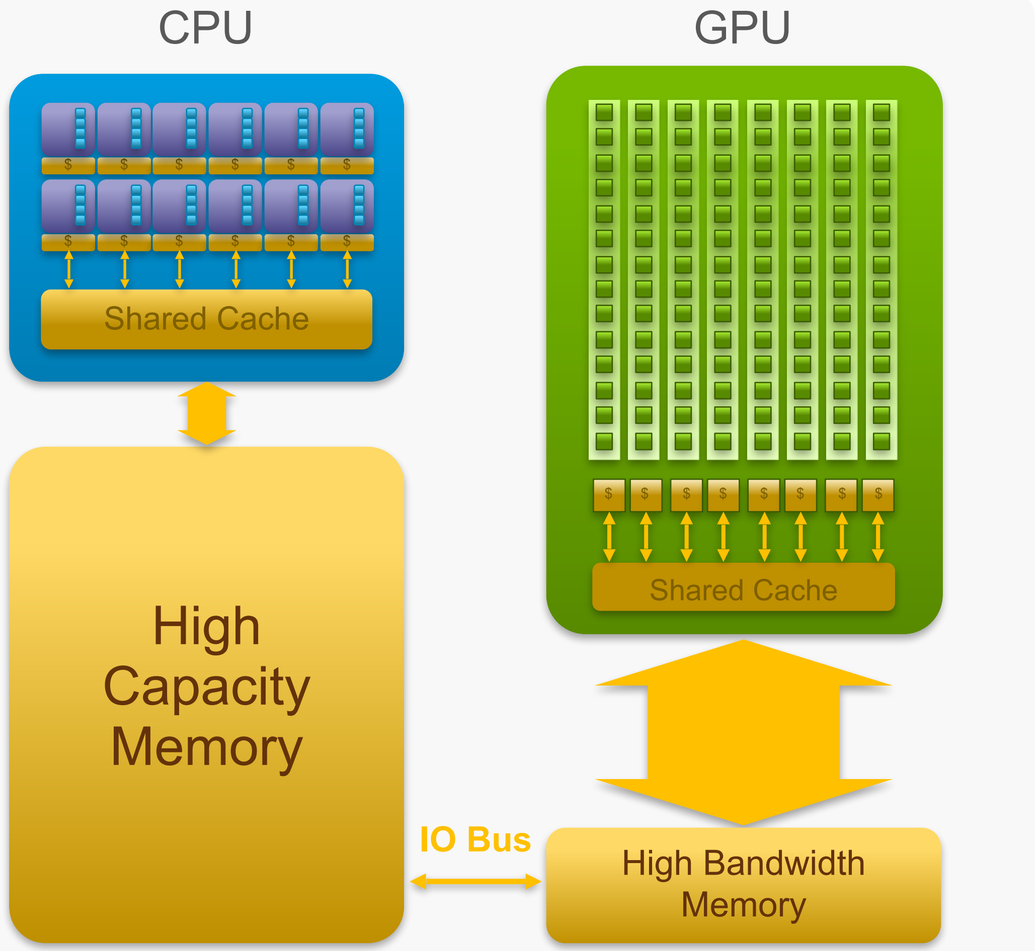
Why do we have to care about data transfers?

Between the host and device

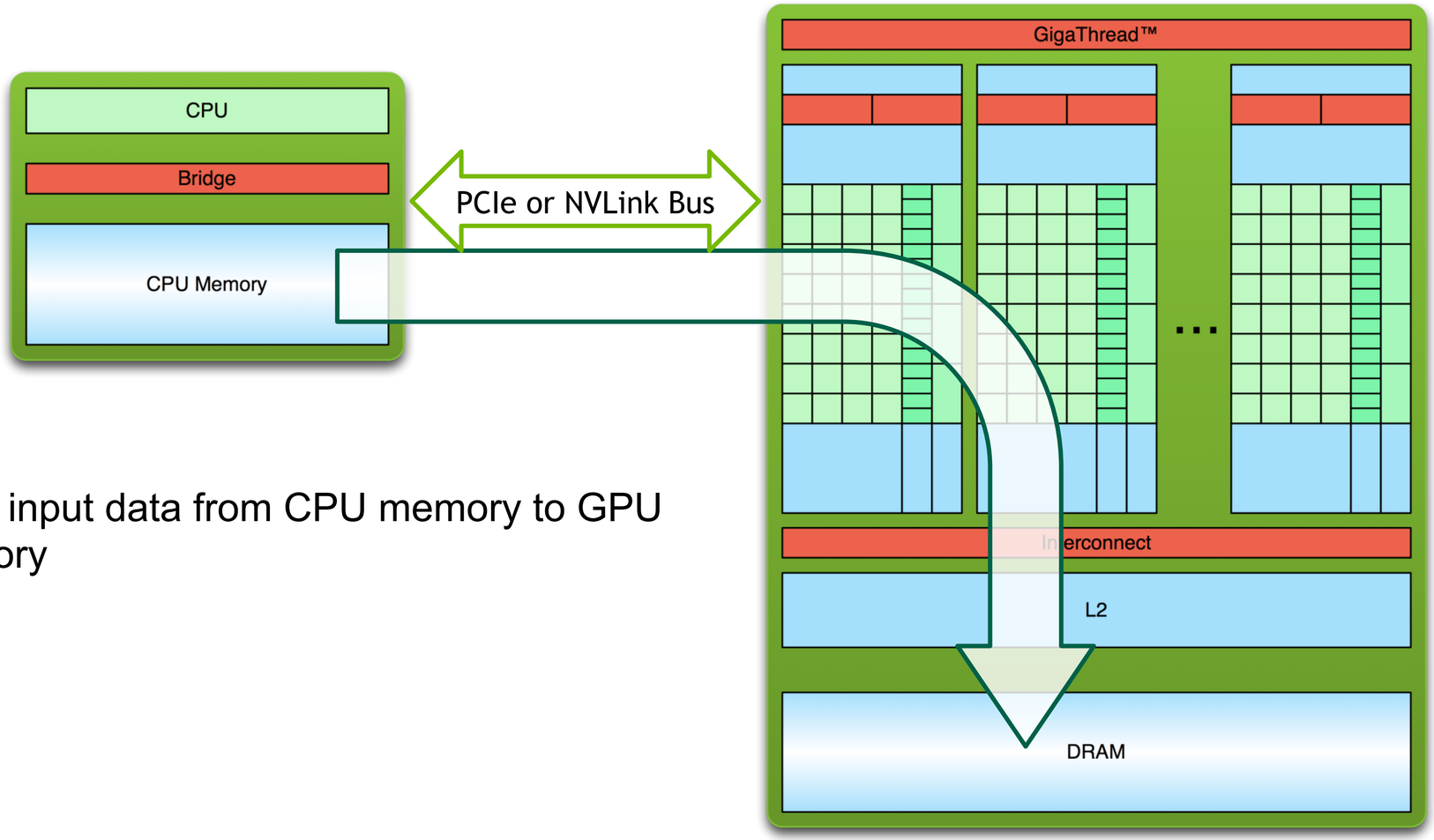
- GPU to its internal memory (HBM2): 900 GB/s
- GPU to CPU via PCIe: 16 GB/s
- GPU to GPU via NVLink: 25 GB/s
- CPU to RAM (DDR4): 128 GB/s

Explicit memory management

- Data must be visible on the device when the kernel is launched
- To maximize performance, data movement needs to be minimise

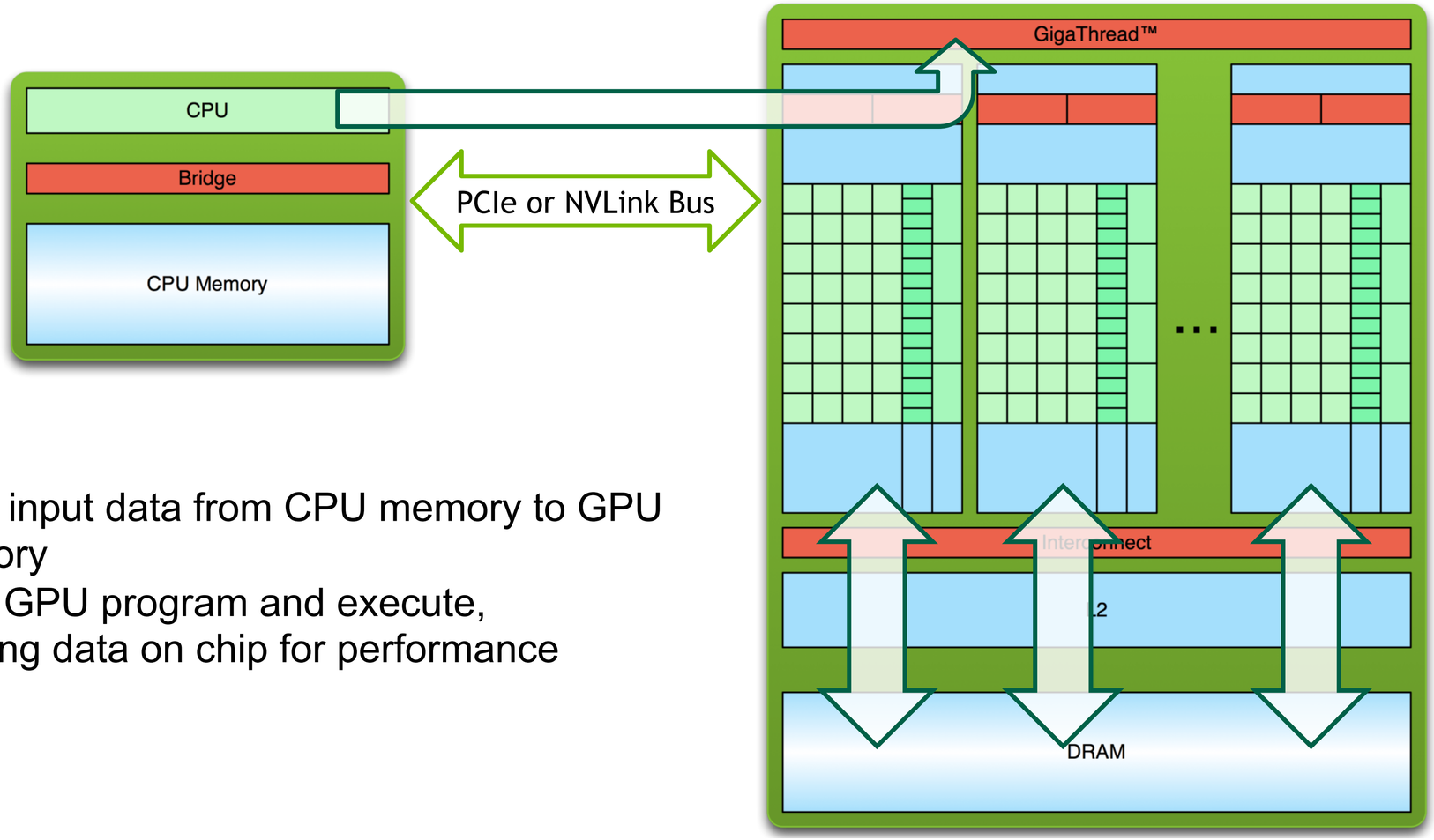


Three simple processing steps



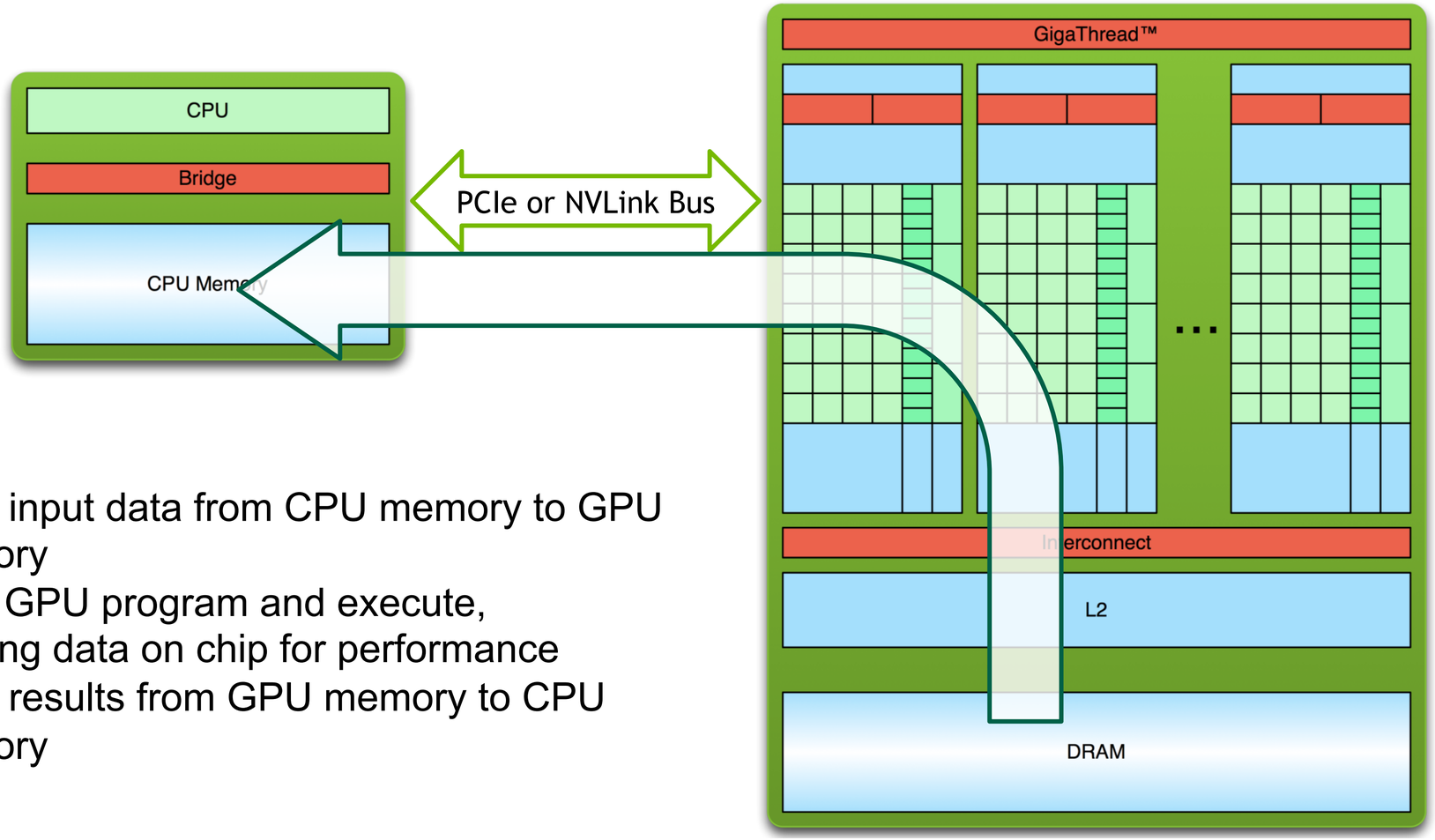
1. Copy input data from CPU memory to GPU memory

Three simple processing steps



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Three simple processing steps

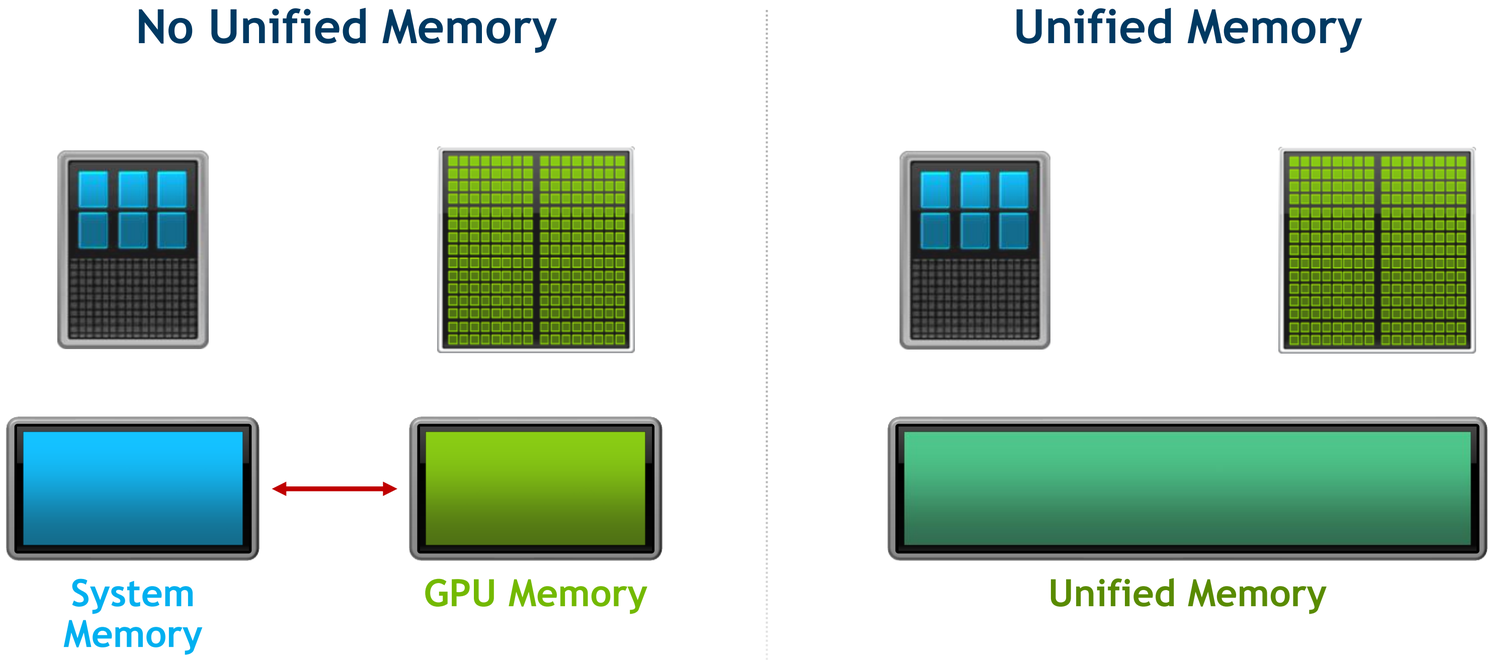


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

CUDA managed memory

Also knows as “Unified Memory”: allocating memory to be accessed on the GPU and the CPU

- Managed memory is accessible to both the CPU and GPU using a single pointer
- The system automatically migrates data allocated in Unified Memory between host and device




```
$ nccx -mp=gpu -gpu=cc80 -Minfo=acc -o binary Code.c
```

```
$ nccx -mp=gpu -gpu=cc80,managed -Minfo=acc -o binary Code.c
```

Excessive data transfers

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

A, Anew resident on host

COPY
HOST / DEVICE

HostToDevice

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

A, Anew resident on accelerator

```
for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
```

```
    ...
}
```

A, Anew resident on host

DeviceToHost

A, Anew resident on accelerator

COPY
DEVICE / HOST

These copies are performed every iteration of the while loop.
NB: in case of two #pragma acc parallel there are 4 copies per while loop iteration.

Compiler often makes a good guess; excessive data transfer

Set export NV_ACC_TIME=1

```
main  NVIDIA  devicenum=0
  time(us): 1,244,792
  49: compute region reached 100 times
      50: kernel launched 100 times
          grid: [65535]  block: [128]
          elapsed time(us): total=52,210 max=572 min=451 avg=522
      50: reduction kernel launched 100 times
          grid: [1]  block: [256]
          elapsed time(us): total=23,058 max=273 min=161 avg=230
  49: data region reached 200 times
      49: data copyin transfers: 900
           device time(us): total=284,383 max=404 min=13 avg=315
      55: data copyout transfers: 900
           device time(us): total=337,661 max=563 min=11 avg=375
  58: compute region reached 100 times
      59: kernel launched 100 times
          grid: [65535]  block: [128]
          elapsed time(us): total=51,338 max=594 min=442 avg=513
  58: data region reached 200 times
      58: data copyin transfers: 800
           device time(us): total=287,305 max=413 min=340 avg=359
      62: data copyout transfers: 800
           device time(us): total=335,443 max=532 min=312 avg=419
```

Compiler often makes a good guess; excessive data transfer

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.04%	652.25ms	1700	383.68us	3.1680us	536.83us	[CUDA memcpy DtoH]
	42.42%	542.07ms	1600	338.79us	309.57us	531.07us	[CUDA memcpy HtoD]
	2.93%	37.409ms	100	374.09us	372.83us	376.73us	main_50_gpu
	2.84%	36.278ms	100	362.78us	360.28us	363.90us	main_59_gpu
	0.77%	9.8475ms	100	98.474us	96.828us	101.12us	main_50_gpu__red
	0.01%	141.28us	100	1.4120us	1.3760us	1.7600us	[CUDA memset]
API calls:	56.07%	962.42ms	4896	196.57us	48.956us	589.45us	cuEventSynchronize

93 % of the time is spent in data transfer

Compiler often makes a good guess

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
         50,   /* blockIdx.x threadIdx.x collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Explicit Data Transfers

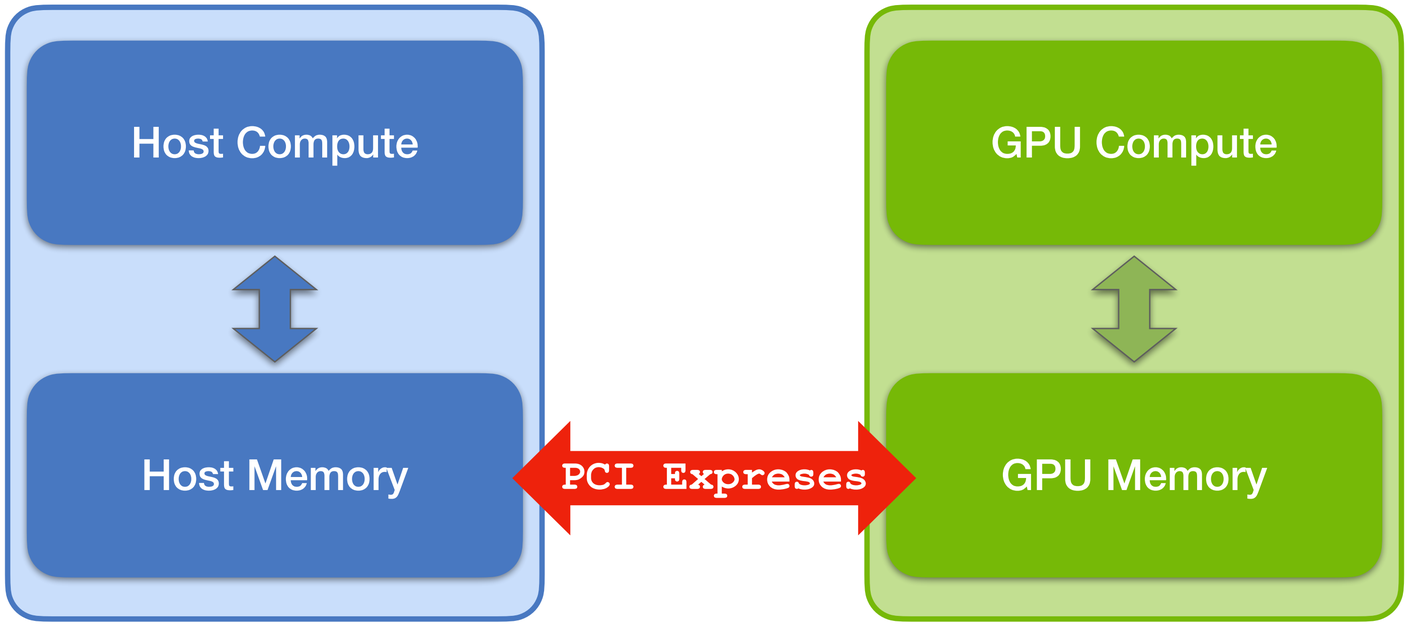
Basic data management

Between GPU <=> CPUs data transferred

- Host = CPU
- Device = parallel accelerator
- If target is multicore hardware, the host and device are the same
- No need to explicitly managed data

If target is multicore hardware

- the host and device are the same, meaning that their memory is also the same
- no need to explicitly manage data when using a shared memory accelerator, such as the multicore target



Managing the data on the device

copy (list) / map(from)	Allocates memory on device and copies data from host to device when entering region and copies data to the host when exiting region
copyin(list) / map(to)	Allocates memory on device and copies data from host to device when entering region
copyout(list) / map(from)	Allocates memory on device and copies data to the host when exiting region
create(list)	Allocates memory on device but does not copy

and delete(list), present(list), present_or_copy[in|out], present_or_create, device_ptr

Structured data directives

a

This **parallel loop** will execute on the **accelerator**, so **a, b, and c** must be visible on the accelerator.

```
#pragma acc parallel loop
for(int i = 0; i < N; I++)
{
     c[i] = a[i] + b[i];

}
```

Structured data directives

```
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
        #pragma acc parallel loop
        for(int i = 0; i < N; i++){
                c[i] = a[i] + b[i];
        }
}
```

Action
[illegible]

Host Memory

Device memory

Shaping arrays

Compiler sometimes cannot determine size of arrays

- Must specify explicitly start/end point
- Memory only exits within the data region
- Must be within a single function

```
/* C/C++ code to offload on the device*/

   #pragma acc data copyin(a[0:N]) copyout(b[s/4:3*s/4])

/* Fortran code to offload on the device*/

   !$acc data copyin(a[1:N]) copyout(b(s/4:3*s/4))
```

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on data, kernels or parallel

Note: data clauses can be used on data, parallel, or kernels

Encompassing Multiple Compute Regions

OpenACC

```
/* A single data region can contain any number of parallel/kernels regions */

void copy (int *A, int *B, int N)
{
  #pragma acc parallel loop copyout(A[0:N]) copyin(A[0:N])
    for (size_t i=0; i<ARRAY_SIZE; i++)
      A[i] = B[i];
}

  #pragma acc data copyout(A[0:N], B[0:N]) copyin(C[0:N])
{
     copy(A, C, N);
     copy(A, B, N);
}
```

Task-3: Add Data clause

02-laplace2d: Data clause with OpenACC

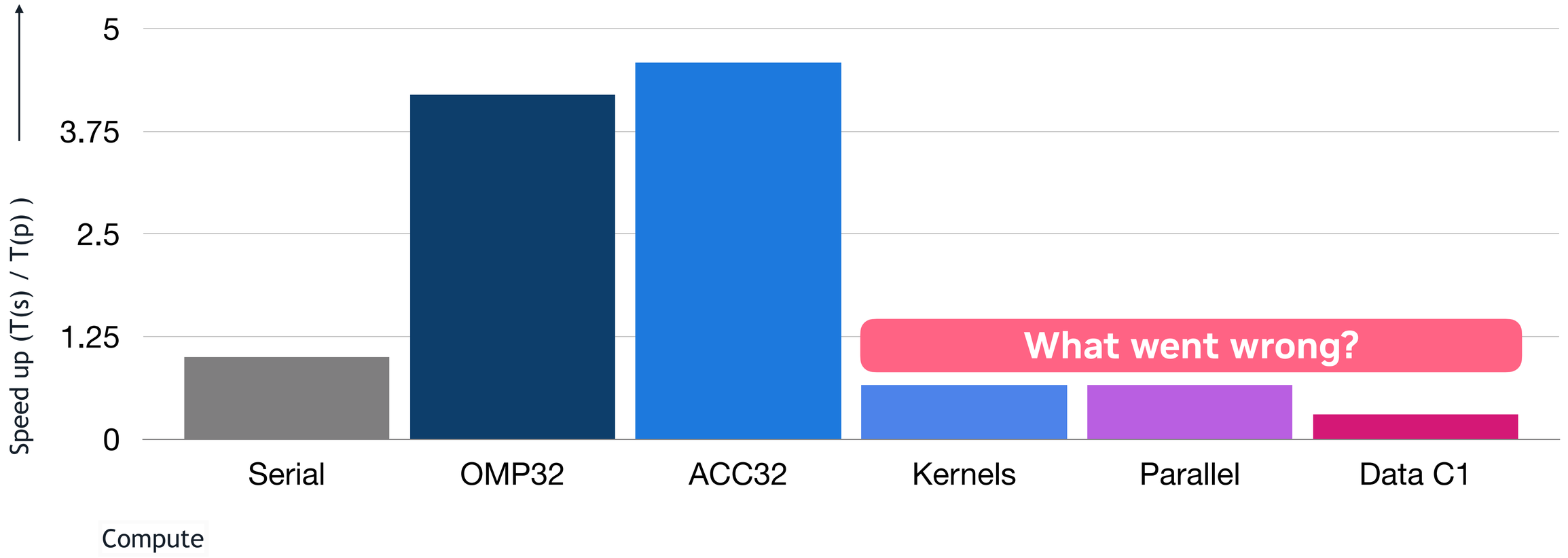
```
while ( error > tol && iter < iter_max ) {
  error=0.0;

for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

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

Things to do

- Make these two loops parallel with OpenACC directives
- Add Parallel construct and data clause
- Try to understand the compiler report to be sure about what the compiler is doing

Performance ~~speed up~~ (higher is better)

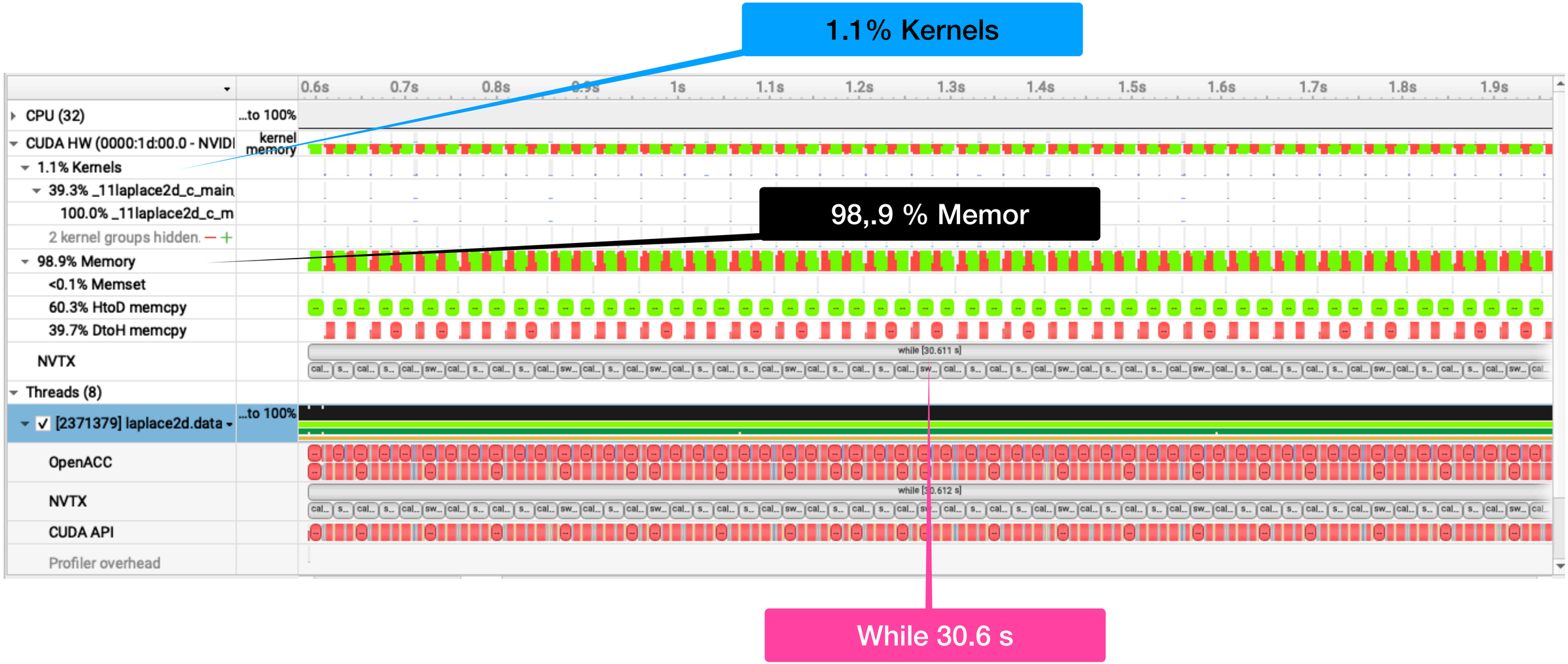
Simulation was performed 1000 Iterations on Leonardo



Look at the report :

compiler often makes a good guess

Nsight system profile



CINECA

02-laplace2d: Data clause with OpenACC

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

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

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

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

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Data clauses provide necessary “shape” to the arrays.

Runtime Breakdown

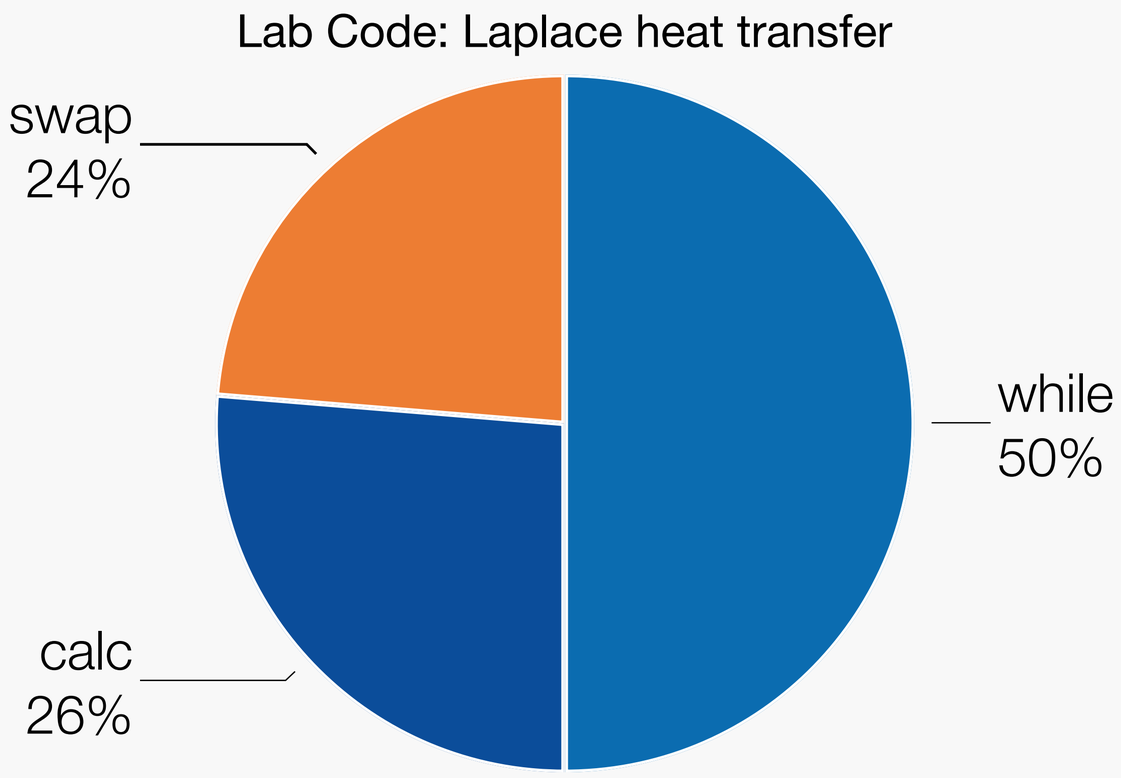
Profiling code

- Obtained information about how the code ran

This includes

- Total runtime
- Runtime of individual routines
- Hardware counters
- Use Profiling tools

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing



Optimize data movement

Managing the data on the device

Data region constructs

- A data region is the dynamic scope of a structured block associated with an implicit or explicit data construct.
- Facilities the sharing of data between multiple parallel regions (kernels, parallel, loop *etc*)
- Must start and end in the scope of the same function or subroutine – it's a structure construct

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

Syntax:

C:

```
#pragma acc data [clause]
{
    code region ...
    including compute related pragmas
}
```

Fortran:

```
!$acc data
    code region ...
    including compute related pragmas
!$acc end data
```

Implied data regions

- Every **kernels** and **parallel** region has an implicit data region surrounding it
- This allows data to exist solely for the duration of the region
- All data clauses usable on a **data** directive can be used on a **parallel** and **kernels** as well

```
#pragma acc kernels copyin(a[0:100])
for(int i = 0; i < N; i++)
{
     a[i] = 0;

}
```

Explicit vs Implicit Data Regions

Explicit — **1 Data Copy**

```
#pragma acc data copyout(a[0:100])
{

 #pragma acc kernels
 { a[I] = I;}

 #pragma acc kernels
 { a[I] = 2 * a[I];}

}
```

Implicit — **2 Data Copies**

```
#pragma acc kernels copyout(a[0:100])
{
  a[I] = I;
}

#pragma acc kernels copy(a[0:100])
{
  a[I] = 2 * a[I];
}
```

The code on the left will perform better than the code on the right.

Encompassing Multiple Compute Regions

OpenACC

```
/* DAXPY in C */
#pragma acc data create( D[0:ARRAY_SIZE], Y[0:ARRAY_SIZE] ) copyin(A) copyout(D[0:ARRAY_SIZE])
{
  #pragma acc loop
    for (size_t i=0; i<ARRAY_SIZE; i++)
      D[i] = 0.0; X[i] = 1.0; Y[i] = 2.0;

  #pragma acc loop
    for (size_t i=0; i<ARRAY_SIZE; i++)
      D[i] = A*X[i] + Y[i];
}
```

Unstructured Data Directives

Unstructured Data Directives

Enter data directive

- Data lifetimes aren't always neatly structured
- The **enter data** directive handles device memory **allocation**
- You may use either the **create** or the **copyin** clause for memory allocation
- The enter data directive is **not** the start of a data region, because you may have multiple enter data directives

```
#pragma acc enter data clauses
< Sequential and/or Parallel code >
#pragma acc exit data clauses
```

```
!$acc enter data clauses
< Sequential and/or Parallel code >
!$acc exit data clauses
```

Managing the data on the device

copyin(list) / map(to)	Allocates memory on device and copies data from host to device when entering region
copyout(list) / map(from)	Allocates memory on device and copies data to the host when exiting region
create(list)	Allocates memory on device but does not copy
delete(list)	Deallocates memory on device without data transfer on exit data

Unstructured Data Directives

There are two unstructured data directives

- **enter data**: Handles device memory allocation, and copies from the Host to the Device. The two clauses that you may use with enter data are create for device memory allocation, and copyin for allocation, and memory copy.
- **exit data**: Handles device memory deallocation, and copies from the Device to the Host. The two clauses that you may use with exit data are delete for device memory deallocation, and copyout for deallocation, and memory copy.

The largest advantage of using unstructured data directives is their ability to branch across multiple functions.

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));
    #pragma acc enter data create(ptr[0:size])
    return ptr;
}

void deallocate(int *ptr)
{
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    #pragma acc parallel loop
    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Unstructured data derivatives

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

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

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

Unstructured Data Directives

With a simple code

Unstructured

```
#pragma acc enter data copyin(a[0:N],b[0:N])\
create(c[0:N])
#pragma acc parallel loop
for(int i = 0; i < N; i++)
   c[i] = a[i] + b[i];

#pragma acc exit data copyout(c[0:N]) \
delete(a,b)
```

- Can have multiple starting/ending points
- Can branch across multiple functions
- Memory exists until explicitly deallocated

Structured

```
#pragma acc enter data copyin(a[0:N],b[0:N])\
copyout(c[0:N])
#pragma acc parallel loop
for(int i = 0; i < N; i++)
   c[i] = a[i] + b[i];

#pragma acc exit data copyout(c[0:N]) \
delete(a,b)
```

- Must have explicit start/end points
- Must be within a single function
- Memory only exists within the data region

Unstructured Data Directives

With a simple code

Be very careful when you create data with unstructured directives.

If you try copying a datum that is PRESENT on the GPU, the copy is not done.

```
< a modified on the host >

#pragma acc enter data copyin(a[0:N]) → host and device copies are out of sync

< a modified on the host > → host and device copies are out of sync

#pragma acc data copyin(a[0:N]) ! copy is ignored

< a used on the GPU > → host and device copies are out of sync
```

Task-4: Structure vs Unstructured data

Data Clause: laplace2d_OpenACC

Analyze the code and refactor the code by following these steps

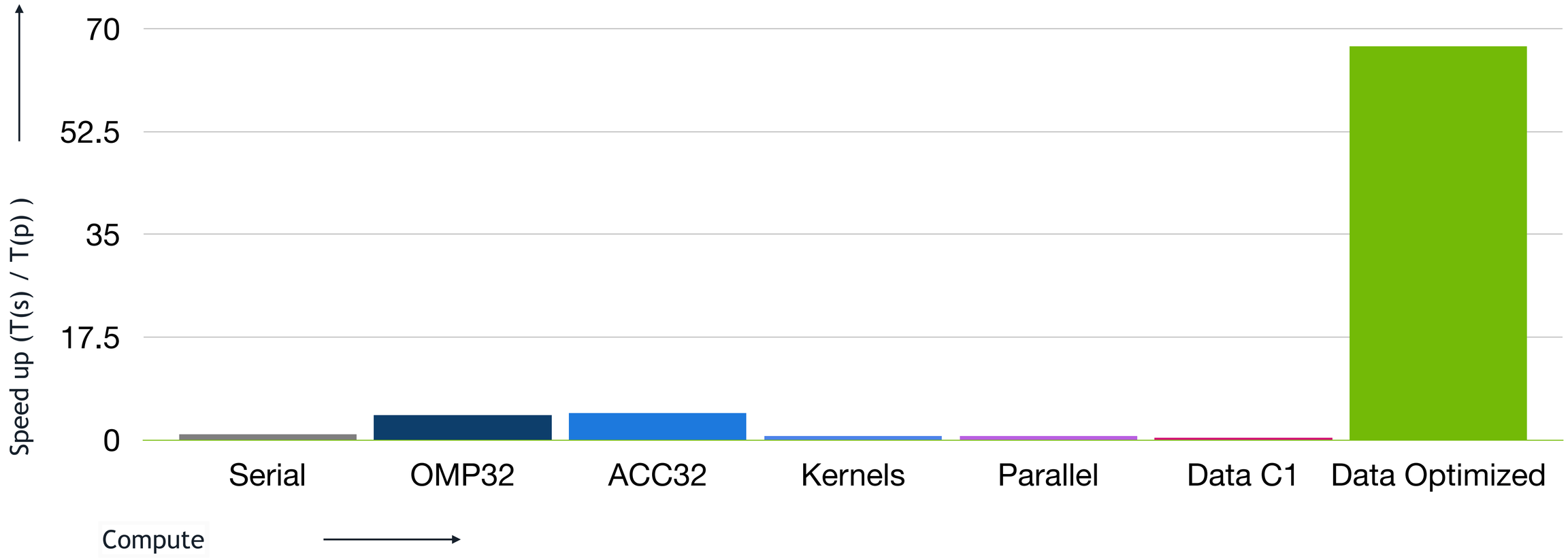
- **Step1:** **Parallelize the code with**
  - Add parallel loop
- **Step2:** **Including data clause in our Laplace code**
  - Compile code with Managed memory
  - Use acc data to minimize transfers (without Managed memory)
  - Add a **structured/ Unstructured data directive** to properly handle the arrays **A** and **Anew**
    Run the Code (With Managed Memory)

Try to understand the compiler report to be sure about what the compiler is doing

- nsys profile -t nvtx,openacc --stats=true --force-overwrite true -o laplace ./laplace

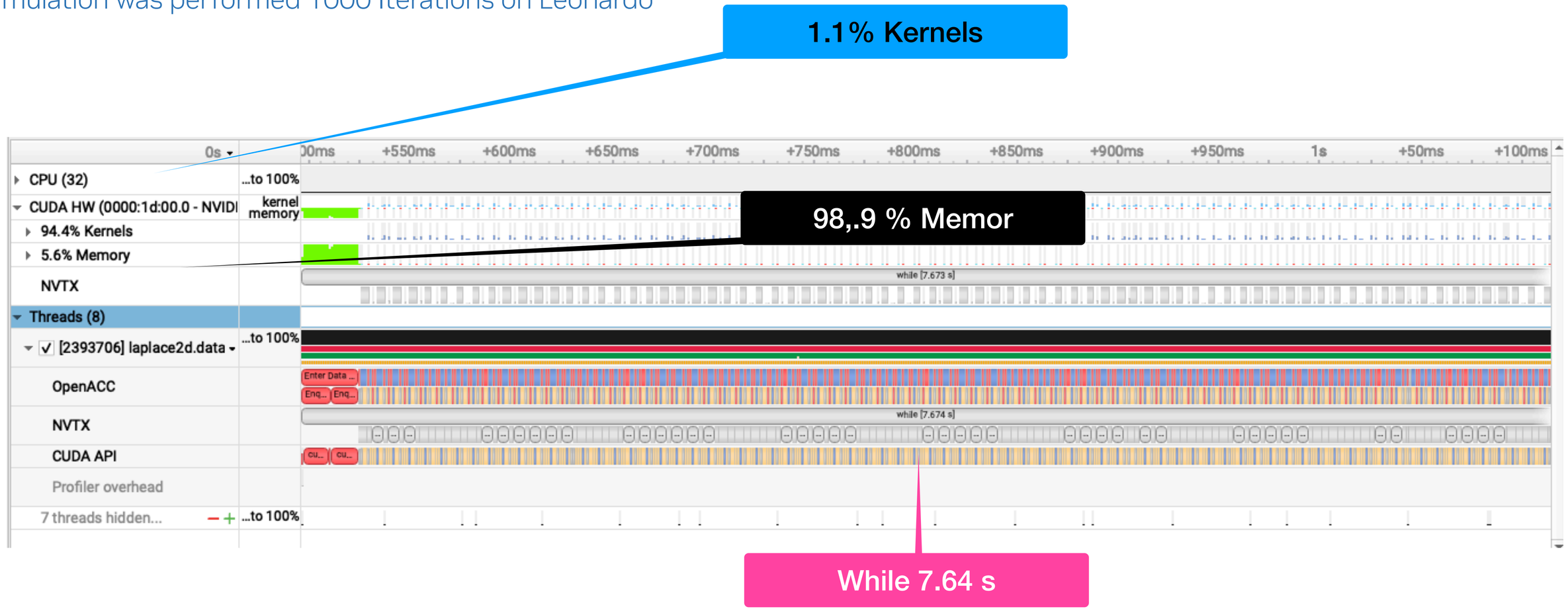
Performance speed up (higher is better)

Simulation was performed 1000 Iterations on Leonardo



Nsight system profile

Simulation was performed 1000 Iterations on Leonardo



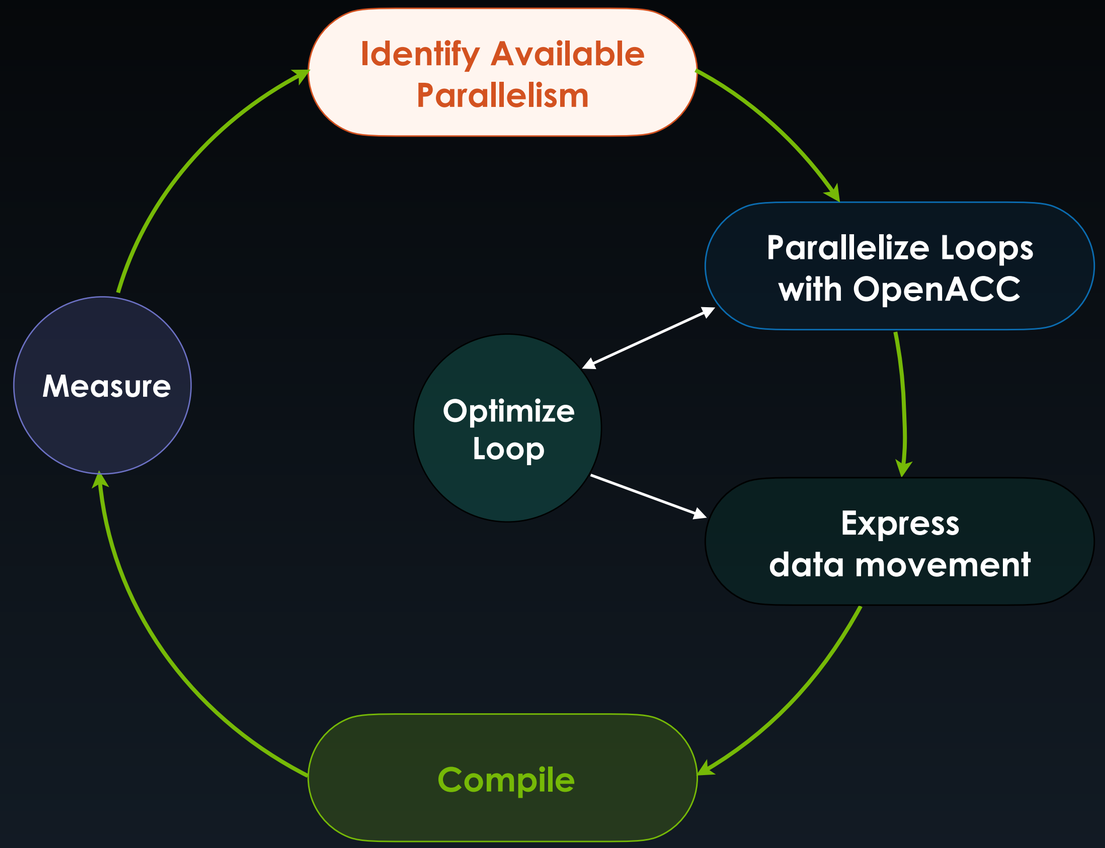
OpenACC Loop Optimisation

OpenX (X = OMP | ACC) porting strategies

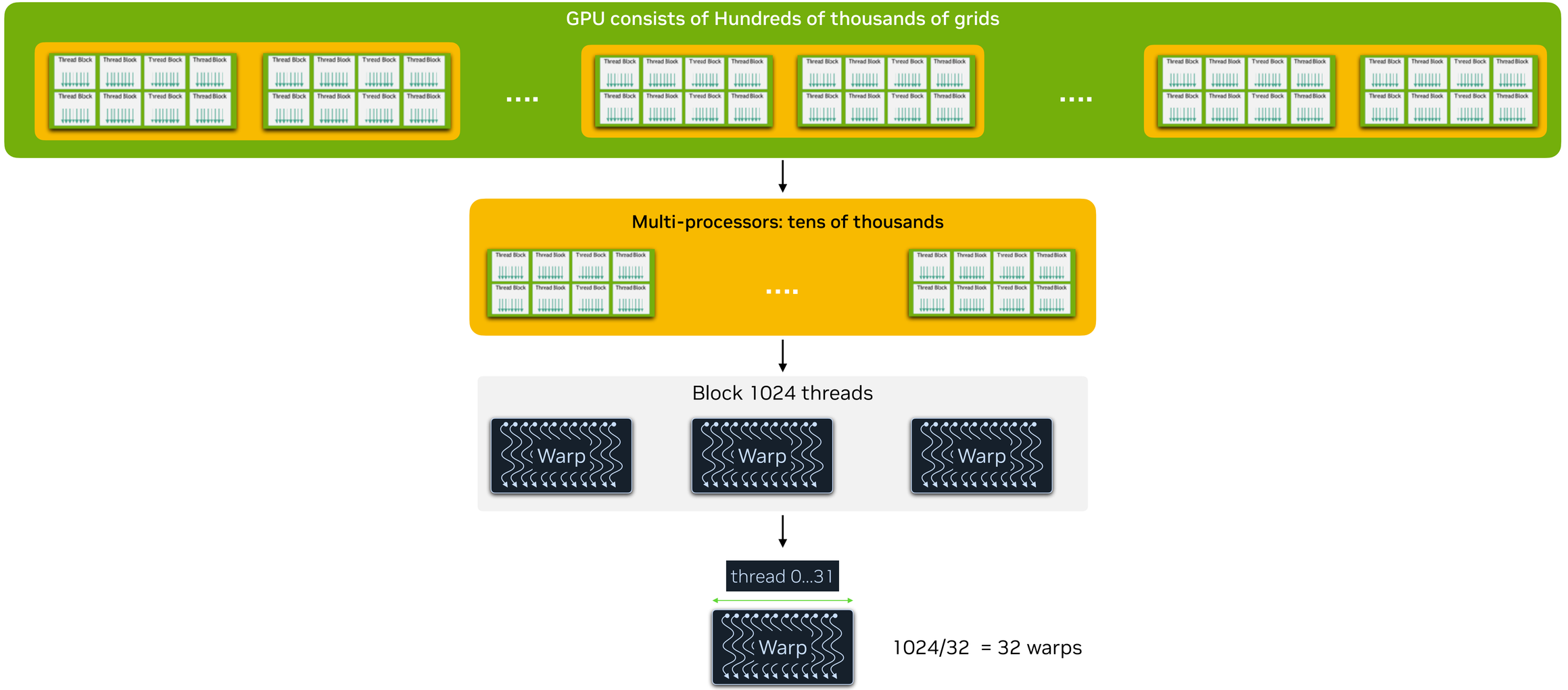
Allows programmers to develop threaded parallel codes on shared memory computational units

Focus on maximising performance

Performance may not increase all-at-once during early parallelisation



GPU Thread hierarchy

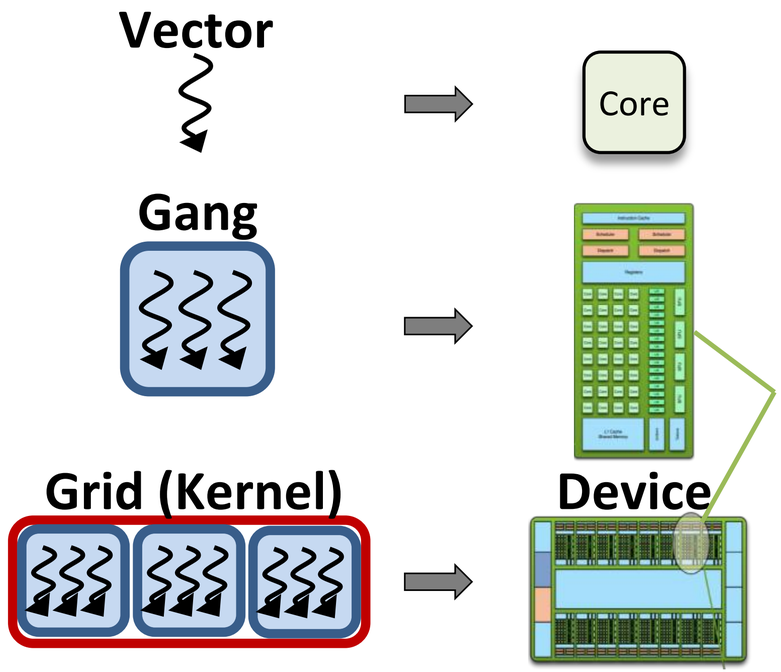


Mapping OpenACC on CUDA model

Possible mapping to CUDA terminology (GPUs) *compiler dependent*

- → gang = block
- → worker = warp
- → vector = threads
  - →Within block (if omitting worker)
  - →Within warp (if specifying worker)

- **Execution Model**



Execution model: three levels of parallelism

PLATFORM	GANG	WORKER	VECTOR
MULTICORE CPU	Entire CPU (NUMA domain)	Core	SIMD vector
MANYCORE CPU (e.g. Xeon Phi)	NUMA domain (whole chip)	Core	SIMD vector
NVIDIA GPU	Thread block	WARP	Thread
AMD GPU	Workgroup	Wavefront	Thread

Number of threads in a gang $N_{threads} = L_{vector} \times N_{workers}$

Gang Worker Vector DEMYSTIFIED

OpenACC
More Science, Less Programming

NVIDIA

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

Gang Worker Vector DEMYSTIFIED

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

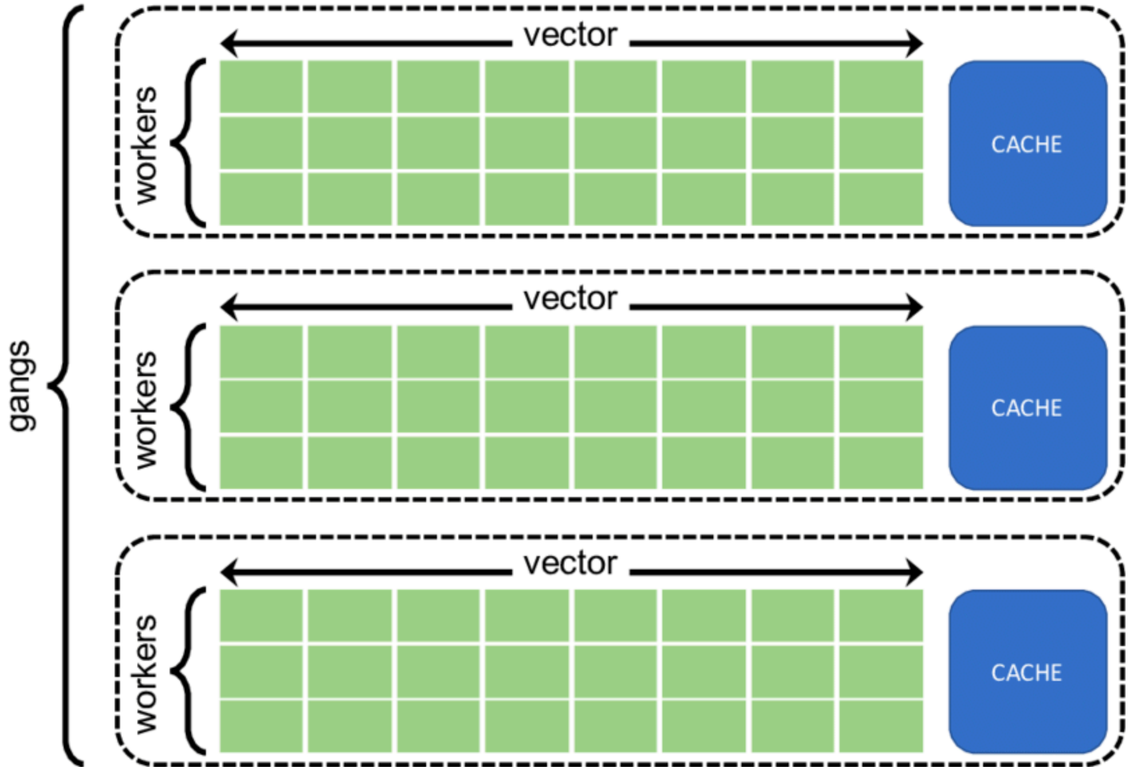
Since gangs operate independently, we can use as many or few as we need.

Gang Worker Vector DEMYSTIFIED

Our painter is like an OpenACC ***worker***, he can only do so much.

His roller is like a ***vector***, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into ***gangs*** to get more done.



OpenACC More Science, Less Programming

NVIDIA

Gang Worker Vector

Gang

- Multiple gangs will be generated, and loops iterations will be spread across the gangs
- Gangs are independent of each other
- There is no way to for the programmer to know exactly how many gangs are running at a given time

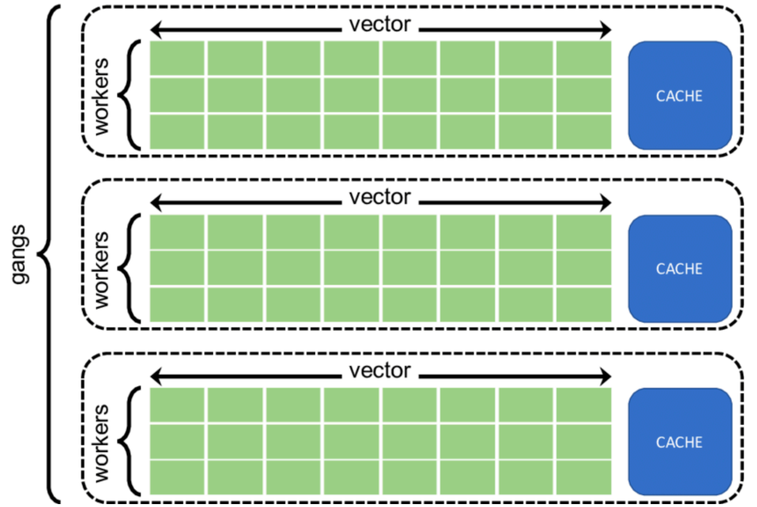
Worker

- To have **multiple vectors** within a gang
- Splits up one large vector into multiple smaller vectors
- Intermediate level between the low-level parallelism implemented in vector and group of threads
- useful when our inner parallel loops are very small, and will not benefit from having a large vector

Vector parallelism:

- Lowest level of parallelism
- Every gang will have at least 1 vector
- Threads work in lockstep (SIMD/SIMT parallelism)

```
#pragma acc loop gang
for ( int i = 0; i < N; i++)
 #pragma acc loop worker
  for ( int j = 0; j < N; j++)
  #pragma acc loop vector
   for ( int k = 0; k < N; k++)
          structured-block
```



Controlling the size of Gang, Worker and Vectors

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses

num_gangs(N)

- Generate N gangs for this parallel region

num_workers(M)

- Generate N gangs for this parallel region

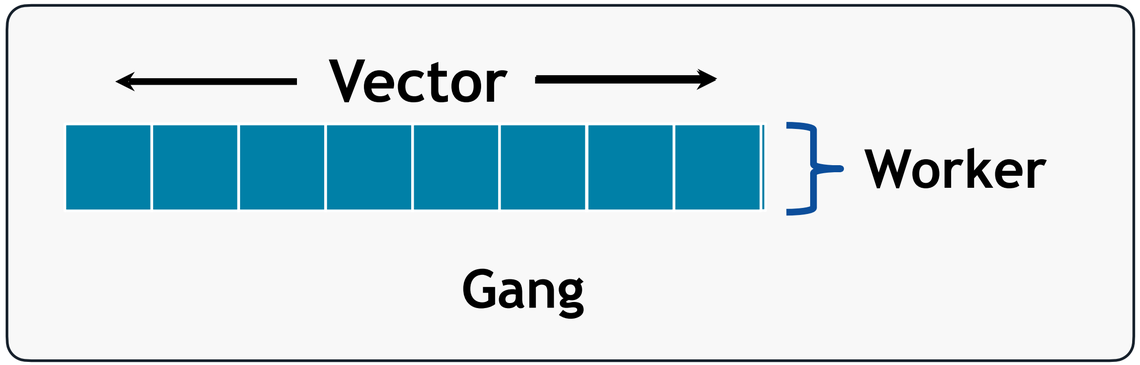
vector_length((P):

- Use a vector length of P for this parallel region

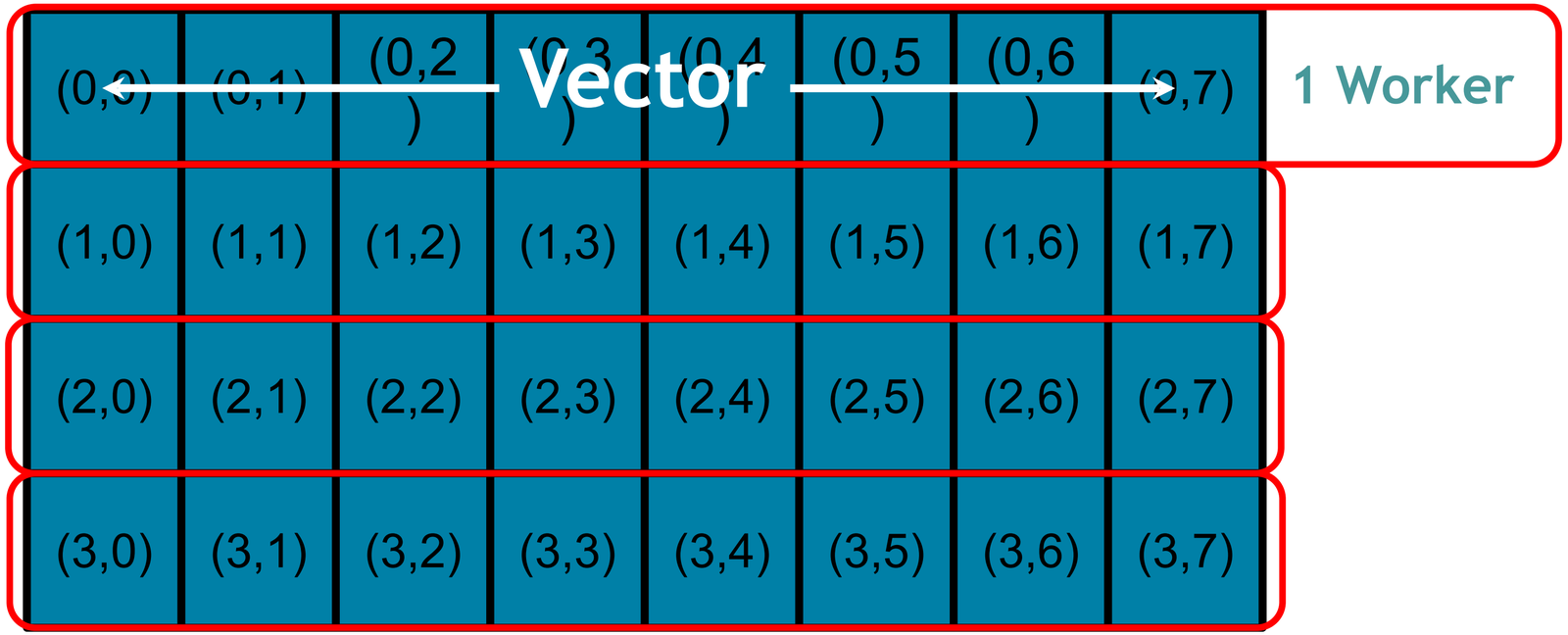
```
#pragma acc paralle nums_gangs(2) num_workers(4) vector_length(32)
 {
  #pragma acc loop worker
   for ( int i = 0; i < N; i++)
     #pragma acc loop vector
     for ( int j = 0; j < N; j++)
            structured-block
 }
```

Rule of 32: general rule of thumb for programming for NVIDIA GPUs is to always ensure that your vector length is a multiple of 32 (which means 32, 64, 96, 128, … 512, … 1024… etc.)

Gang Worker Vector DEMYSTIFIED



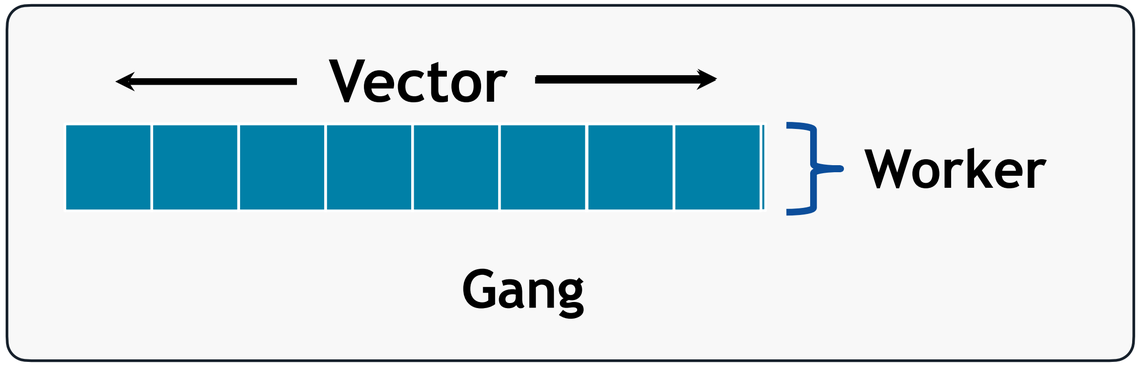

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```



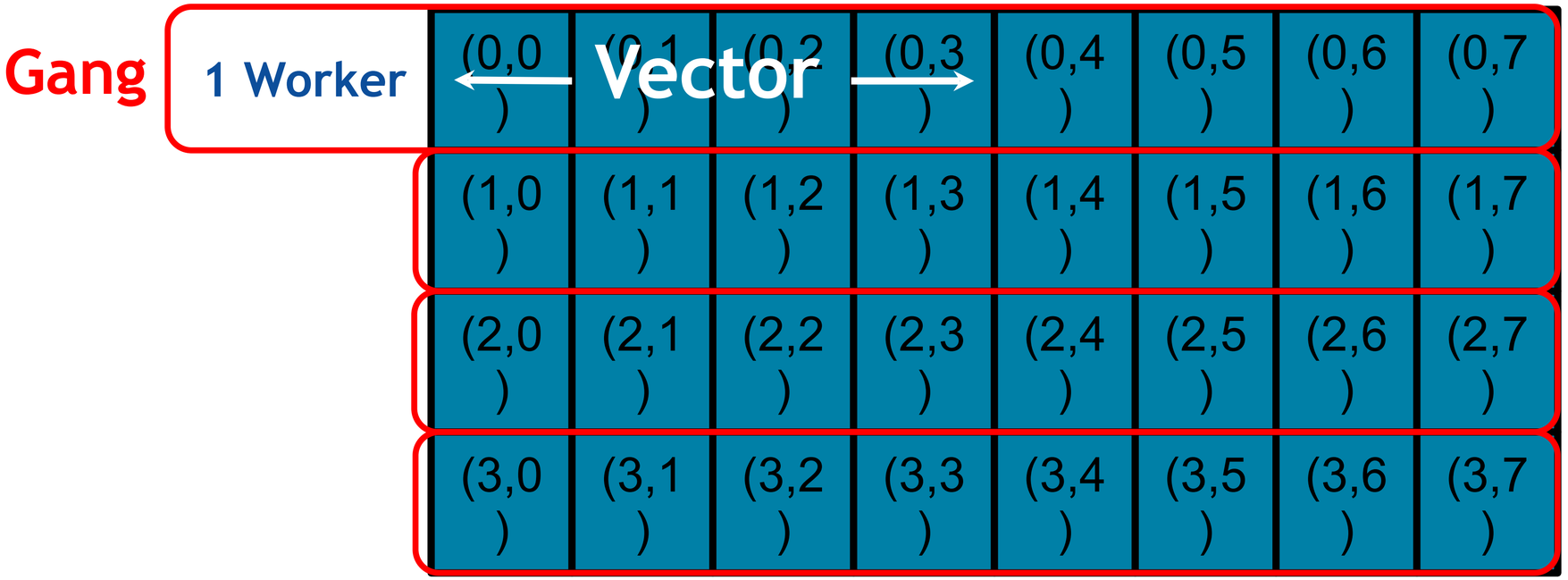
Gang

- The vectors are colored, so that we can observe which loop iterations they are being applied to
- Based on the size of this loop nest, the compiler will (theoretically) generate **4 gangs**

Gang Worker Vector DEMYSTIFIED

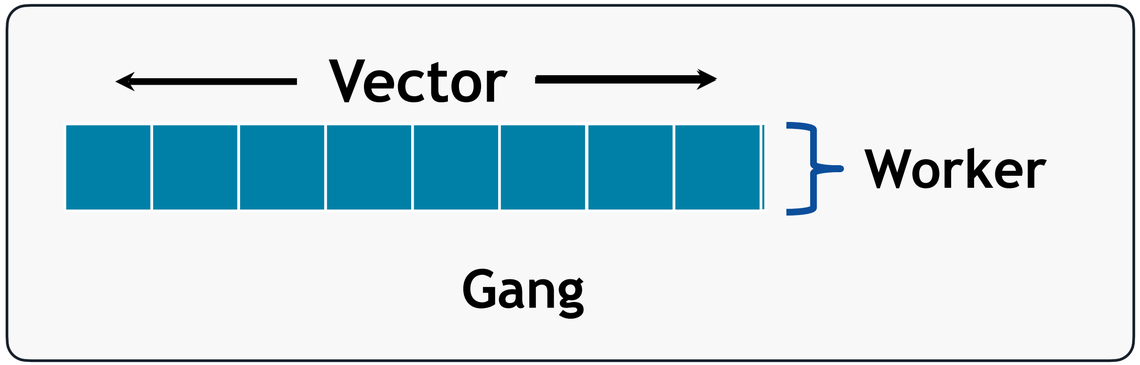



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```

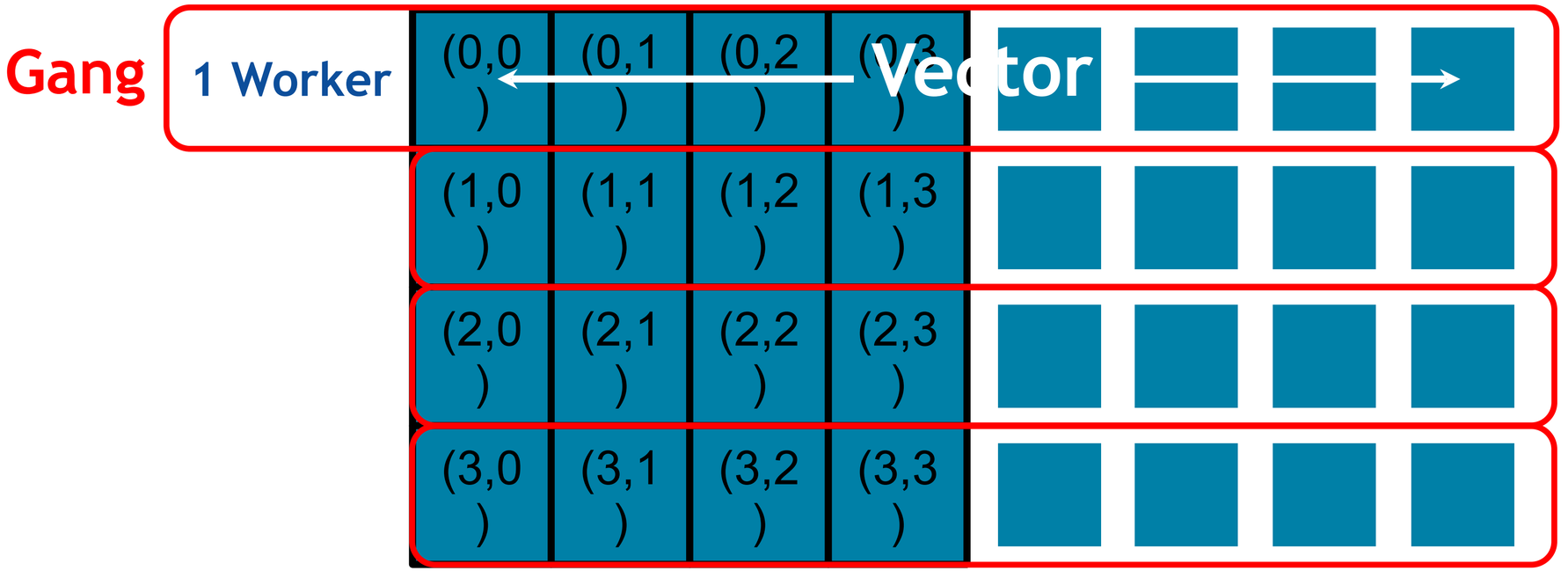


- We are still generating 4 gangs, but now each vector is computing two loop iterations
- If we wanted to generate **more gangs**, we would need to increase the size of the outer-loop

Gang Worker Vector DEMYSTIFIED

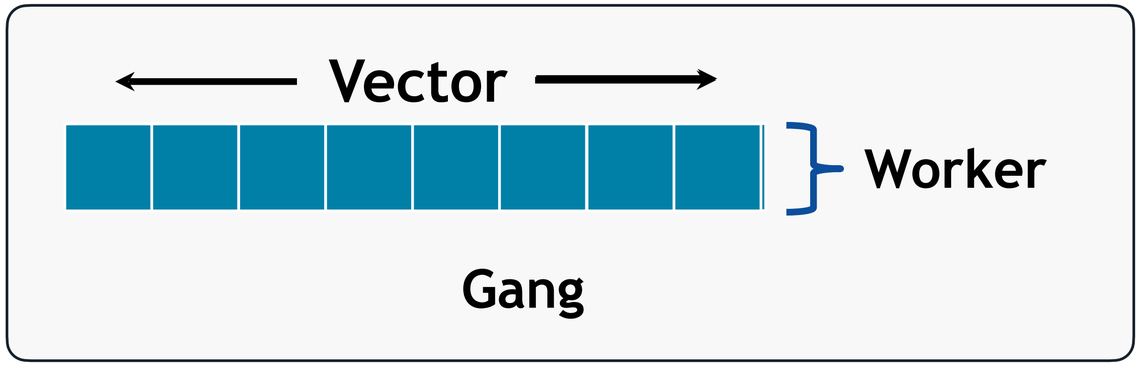



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 4; y++){
        array[x][y]++;
     }
}
```



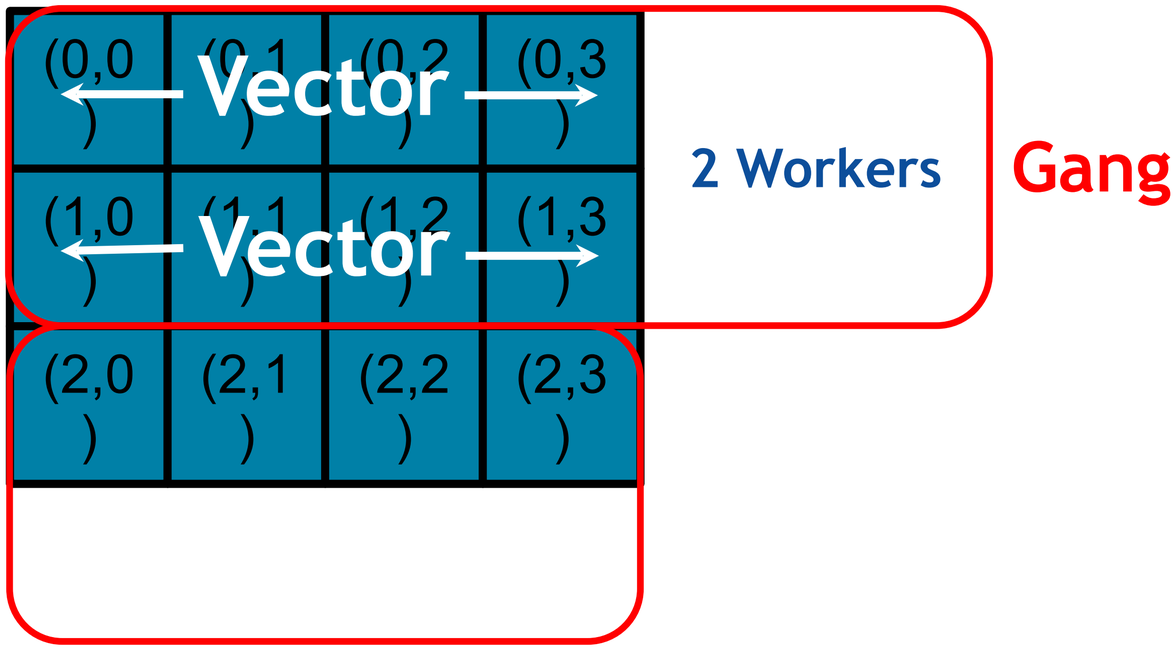
- We can see that our vector length is **much larger** than our inner-loop
- We are **wasting** half of our vector, meaning our code is performing half as well as it could

Gang Worker Vector DEMYSTIFIED



We can fix this by **breaking our vector** up among **2 workers**

```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 4; y++){
        array[x][y]++; }
}
```



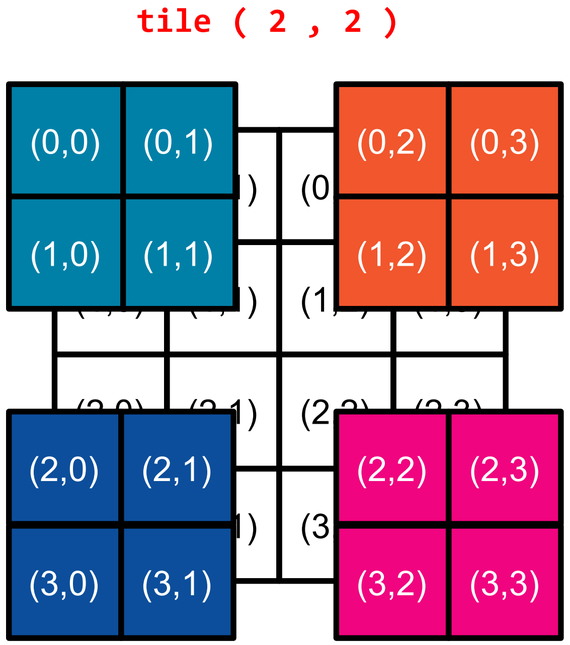
- We are no longer wasting a portion of our vectors, since the smaller vector size now fits our loop properly
- We always need to consider the size of the loop when choosing the gang worker vector dimensions

The Tile clause

tile(x, y, z . . . .)

- Breaks multidimensional loop into “blocks ”
- Can increase data locality in some codes
- Will be able to execute multiple tiles simultaneously

```
#pragma acc kernels loop tile(2,2)
for ( int i = 0; i < N; i++)
    for ( int i = 0; i < N; i++)
     structured-block
```



The Tile clause

- Collapse(n): Applies the associated directive to the following n tightly nested loops
- Useful when loop extents are short
- The compiler may decide to collapse loops anyway, check the report!

```
#pragma acc parallel
#pragma acc loop collapse(2)
for ( int i = 0; i < N; i++)
    for ( int i = 0; i < N; i++)
       structured-block
```

```
#pragma acc parallel
#pragma acc loop
for ( int ij = 0; ij < N*N; ij++)
    i = ij/N;
    j = ij/N;
       structured-block
```

OpenACC collapse clause

Without reduction

- The inner-most loop is not parallelizable
- Multithreads could attempt to write to c[i][j]
- It cannot be guarantee that the threads will do the reduction properly
- Erroneous results

```
for (int i = 0; i < size; i++)
  for (int j = 0; j < size; j++)
    #pragma acc parallel loop
      for (int k = 0; k < size; j++)
        c[i][j] += a[i][j] + b[i][j];
```

OpenACC Reduction construct

Compiler is often very good at detecting when reduction is needed

With reduction clause

- Takes many values and “reduces” to a single value
- Each thread calculates its part
- Perform a partial reduction on the loop iterations the they compute
- After the loop, the compiler will perform a final reduction to produce a single result using the specified operation

```
for (int i = 0; i < size; i++)
  for (int j = 0; j < size; j++)
  #pragma acc parallel loop reduction(+:tmp)
       for (int k = 0; k < size; j++)
          tmp += a[i][j] + b[i][j];
          c[i][j] = tmp;
```

OpenACC Reduction construct

Compiler is often very good at detecting when reduction is needed

Operator	Description	Example
`+`	Addition/Summation	`reduction(+:sum)`
`*`	Multiplication/Product	`reduction(*:product)`
`max`	Maximum value	`reduction(max:maximum)`
`min`	Minimum value	`reduction(min:minimum)`
`&`	Bitwise and	`reduction(&:val)`
`|`	Bitwise or	`reduction(|:val)`
`&&`	Logical and	`reduction(&&:val)`
`||`	Logical or	`reduction(||:val)`

Independent clause

Independent Clause: A way for programmer to guarantee to the compiler that a loop is parallelizable

- Use carefully
- Overrides compiler analysis for dependence
- In a kernel construct the loop independent tells to the compiler that the items in the loop are not containing any dependence on each other

```
#pragma acc kernels loop
independent
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

```
#pragma acc kernels loop
for ( int i = 0; i < n; i++)
          < Parallel Loop >

#pragma acc independent
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

```
#pragma acc kernels
#pragma acc loop independent
for ( int i = 0; i < n; i++)
          c[i] = 2.*c[m+i];
```

m>n

Auto Clause: more-or-less the opposite of the Independent

Auto clause

Auto Clause: more-or-less the opposite of the Independent

- the compiler will trust anything that the programmer decides
- the compiler will double check the loops, and decide whether or not to parallelize them.

```
#pragma acc kernels loop auto
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

SEQ clause

SEQ (sequential) clause tells the compiler to run the loop sequentially on the parallel hardware

- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially
- The compiler may automatically apply the seq clause to loops as well, if it knows the count is short, or contains a loop-carried dependency

```
#pragma acc parallel loop
  for (int i = 0; i < size; i++)
  #pragma acc loop
    for (int j = 0; j < size; j++)
      #pragma acc loop seq
        for (int k = 0; k < size; j++)
          c[i][j] += a[i][j] + b[i][j];
```

Private and Firstprivate clause

The private clause allows the programmer to define a list of variables as “thread-private”

- Each thread will be given a private copy of every variable win the comma separated list
- **Firstprivate** is like private except that the private values are initialised to the same value used on the host, **Private** variable are not initialised

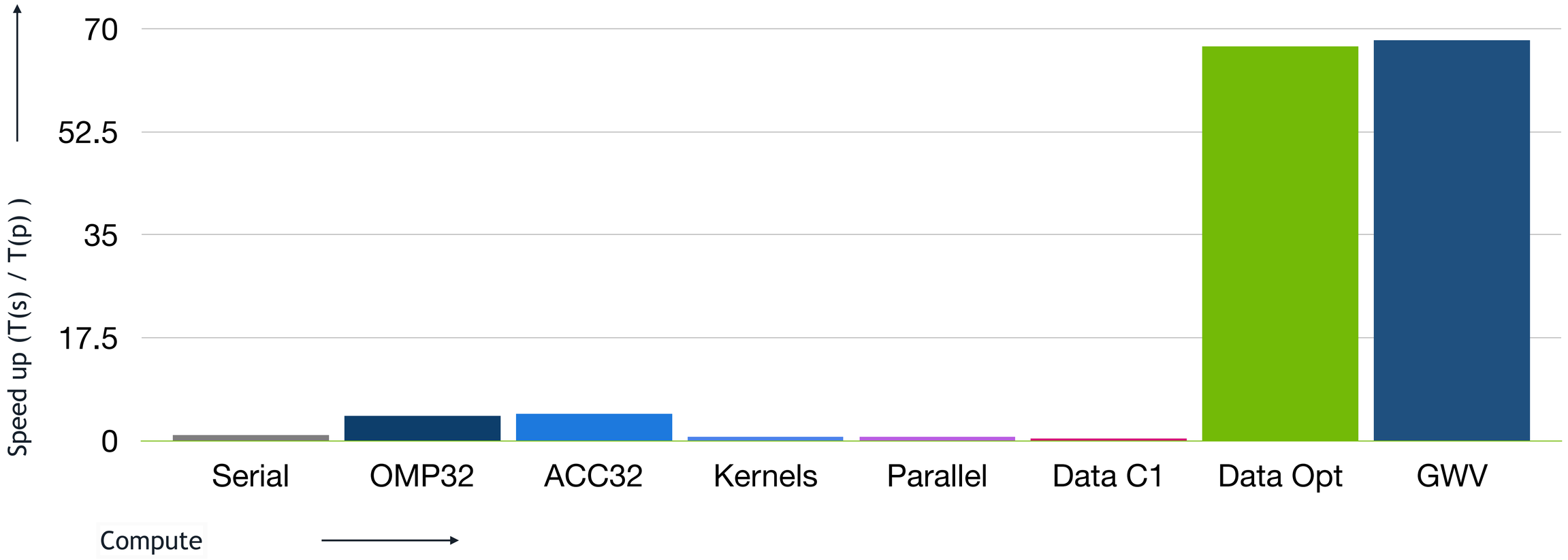
```
double tmp[3]
 #pragma acc kernels loop privarte(tmp[0:3]
  for (int i = 0; i < size; i++)
    tmp[0] += <value>;
    tmp[1] += <value>;
    tmp[2] += <value>;
```

! Host value of ‘tmp’ remains unchanged

Task -5 : Optimise the loop

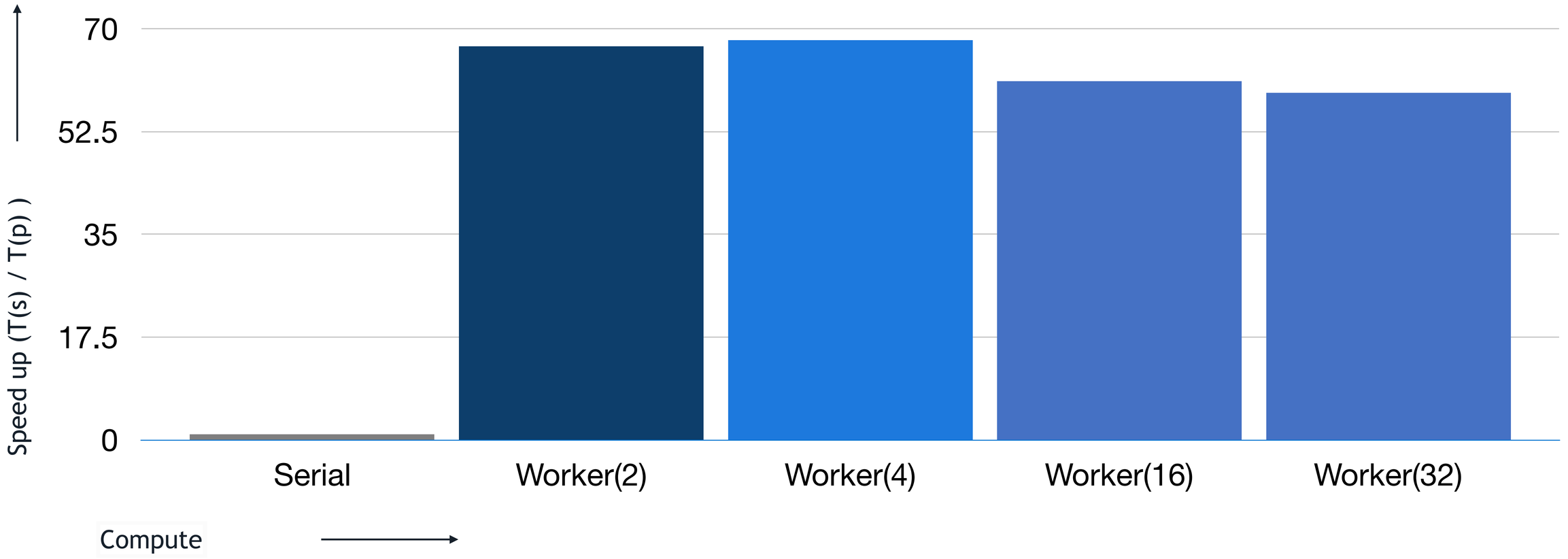
Performance speed up (higher is better)

Simulation was performed 1000 Iterations on Leonardo



Performance speed up (higher is better)

Simulation was performed 1000 Iterations on Leonardo



CINECA

Ministero dell'Università e della Ricerca

EuroHPC Joint Undertaking

Regione Emilia-Romagna

Atos

INFN

Nitin Shukla
Scientific Application Engineer

Mail: n.shukla@cineca.it

SPACE

Thank you for your attention!