Introduction to Parallel and Distributed Programming

Unit 7 – Introduction to General Purpose Graphics Processing Unit (GPGPU) programming

Pablo Arias

slides by Ricard Borrell, Sergi Laut

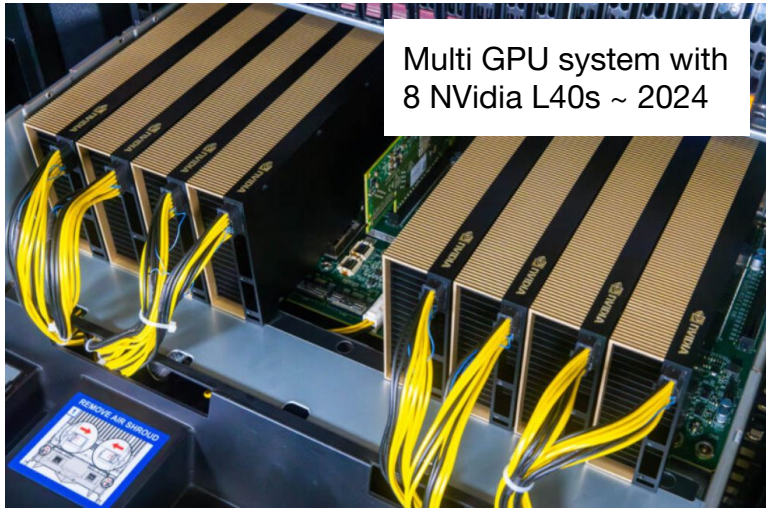
History GPU programming

NVidia GForce 4 ~2002

GPUs were originally designed to process images quickly, having many simple ALUs that perform simple operations on pixels.

- ~ 2002 first attempts to use GPUs for basic scientific computing.
- ~ 2006 CUDA programming language (C-like lang)
- ~ 2009 OpenCL open source version CUDA.
- 2012 OpenACC
- 2013 OpenMP 4.0 with GPU support

Today there many other approaches to GPU programming (other CUDA and OpenACC).

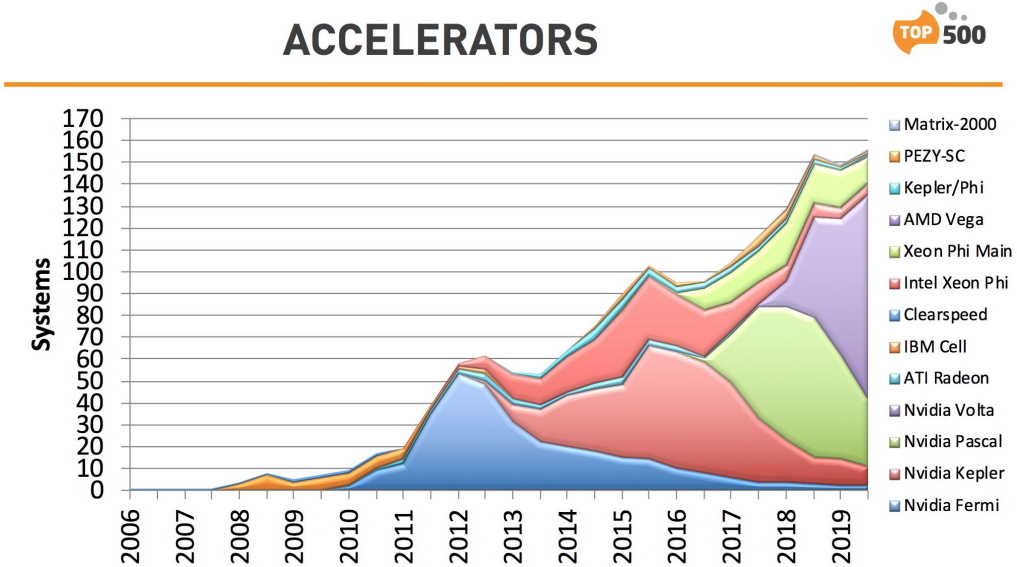


Multi GPU system with 8 NVidia L40s ~ 2024

Motivation

GPUS have become the processor of choice for many types of intensively parallel computations.

- Gaming,
- Deep Learning and AI,
- Data Science,
- Autonomous Driving,
- crypto, etc



On the leading edge Supercomputing sector, the GPUs adoption is growing due to its compute density and high flop/watt ratio

Image Left: https://www.hpcwire.com/2018/03/27/nvidia-riding-high-as-gpu-workloads-and-capabilities-soar/
Image Right: top500.org

References

Rob Farber. 2016. **Parallel Programming with OpenACC** (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Juckeland, Guido & Chandrasekaran, Sunita. (2017). **OpenACC for Programmers: Concepts and Strategies**.

David B. Kirk and Wen-mei W. Hwu. 2016. **Programming Massively Parallel Processors: A Hands-on Approach** (3rd. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

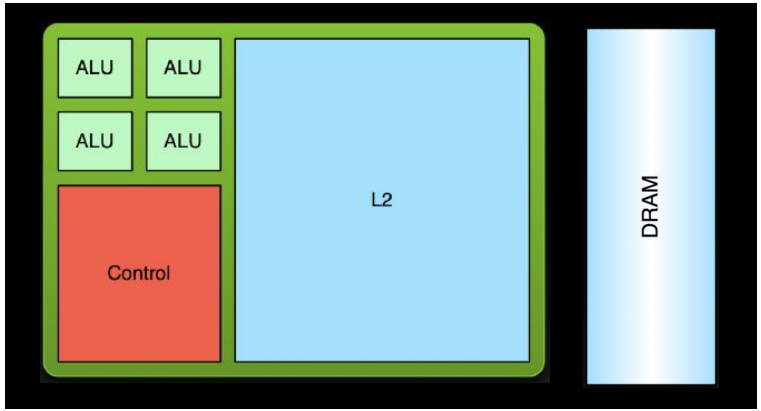
openAcc.org → tutorials, specification

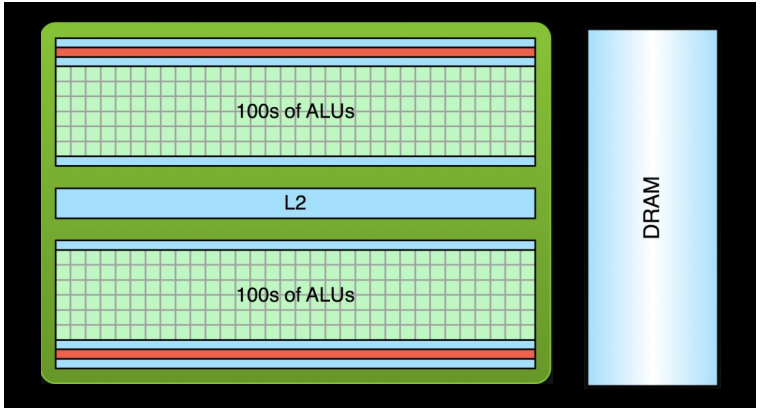
Outline

1. GPGPU computing
2. GPU Architecture outline
3. OpenACC in a basic example
4. OpenACC Data Management
5. Concluding remarks

GPGPU computing

Low Latency vs High Throughput





CPU:
Low latency: make each single thread fast.

Optimized for **low-latency** access to cached data.

Control logic for out-of-order and speculative execution.

GPU:
Optimized for data-parallel computations

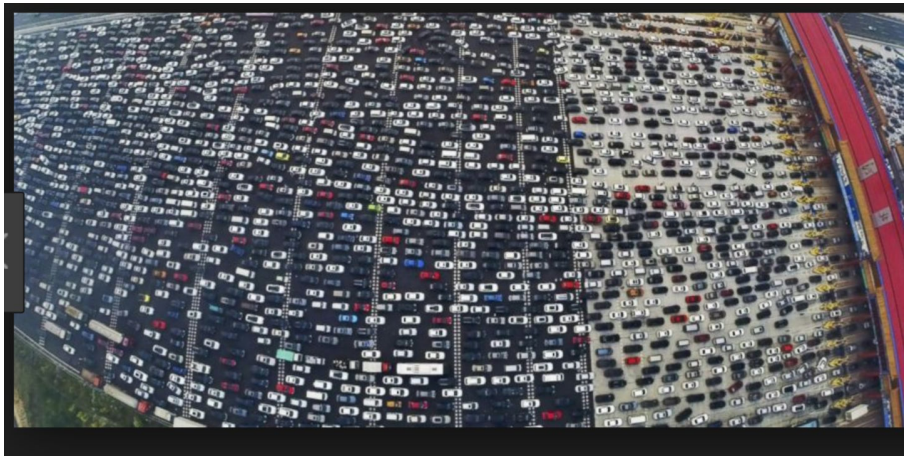
Throughput matters more than single threads

Tolerance to latency: hide memory latency through parallelism. Context switching: if a thread is waiting for data, it's quick to switch to another thread.

More transistors dedicated to computation

Credit: Cliff Woolley, NVIDIA

Low Latency vs High Throughput



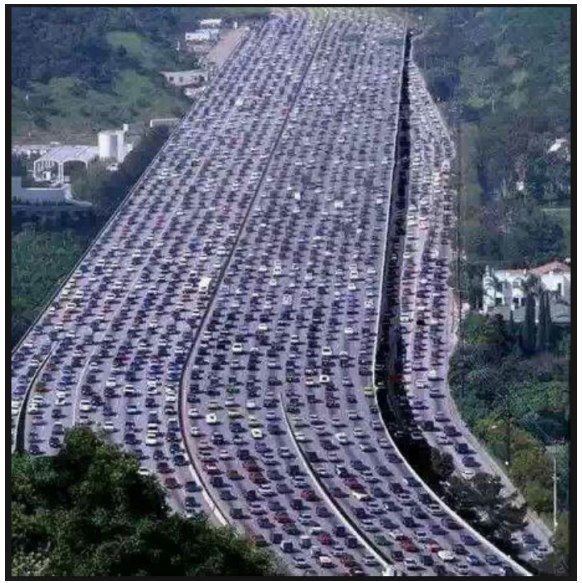
Throughput conditions

Which are the conditions to achieve high throughput?

Volume: many computations

Regularity: single instruction applied to different data (SIMD)

Avoid data contentions and serialization



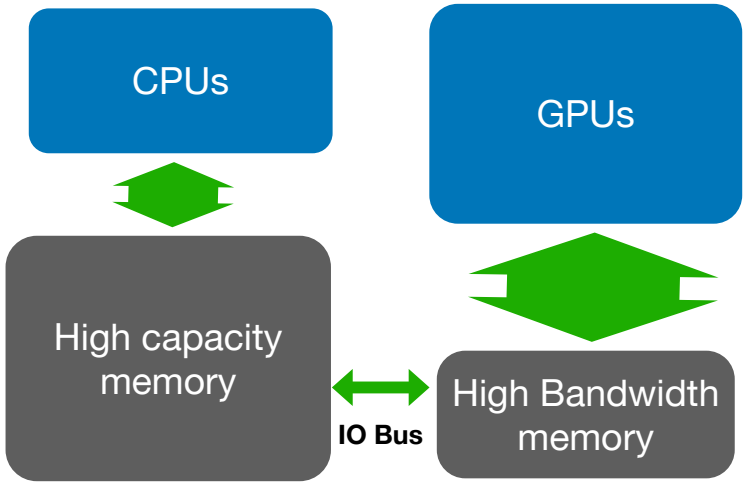
Host/device model

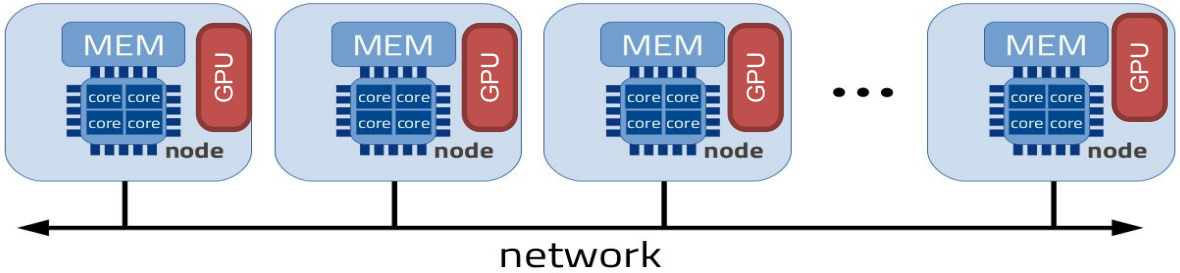
CPU host controls the device and manage inter-node communications.

CPU & GPU can process in parallel.

CPUs for sequential or irregular workloads where latency matters.

GPUs for parts where throughput wins.





Memory bandwidths

Some figures:

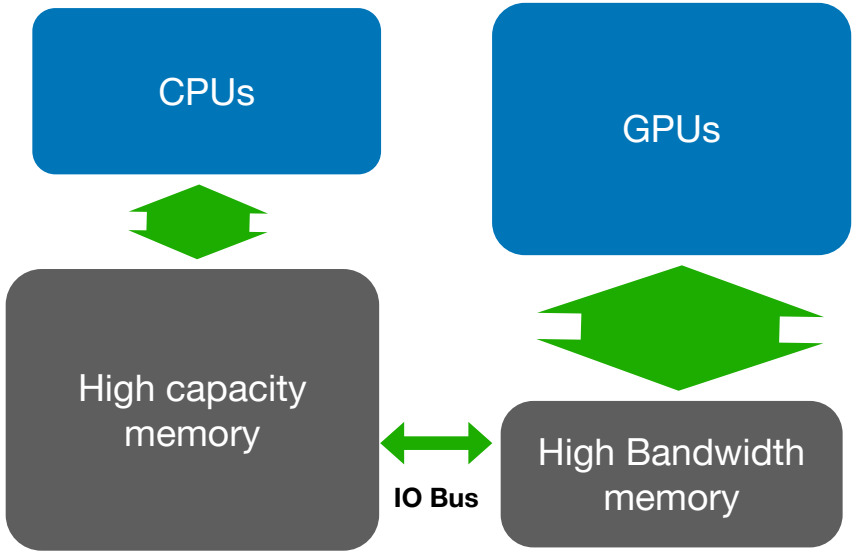
GPU memory

```
RTX 1080 Ti:   11GB - GDDR5   484GB/s
RTX 4090    :   24GB - GDDR6X    1TB/s
H100        :   80GB - GDDR6   3.4TB/s
```

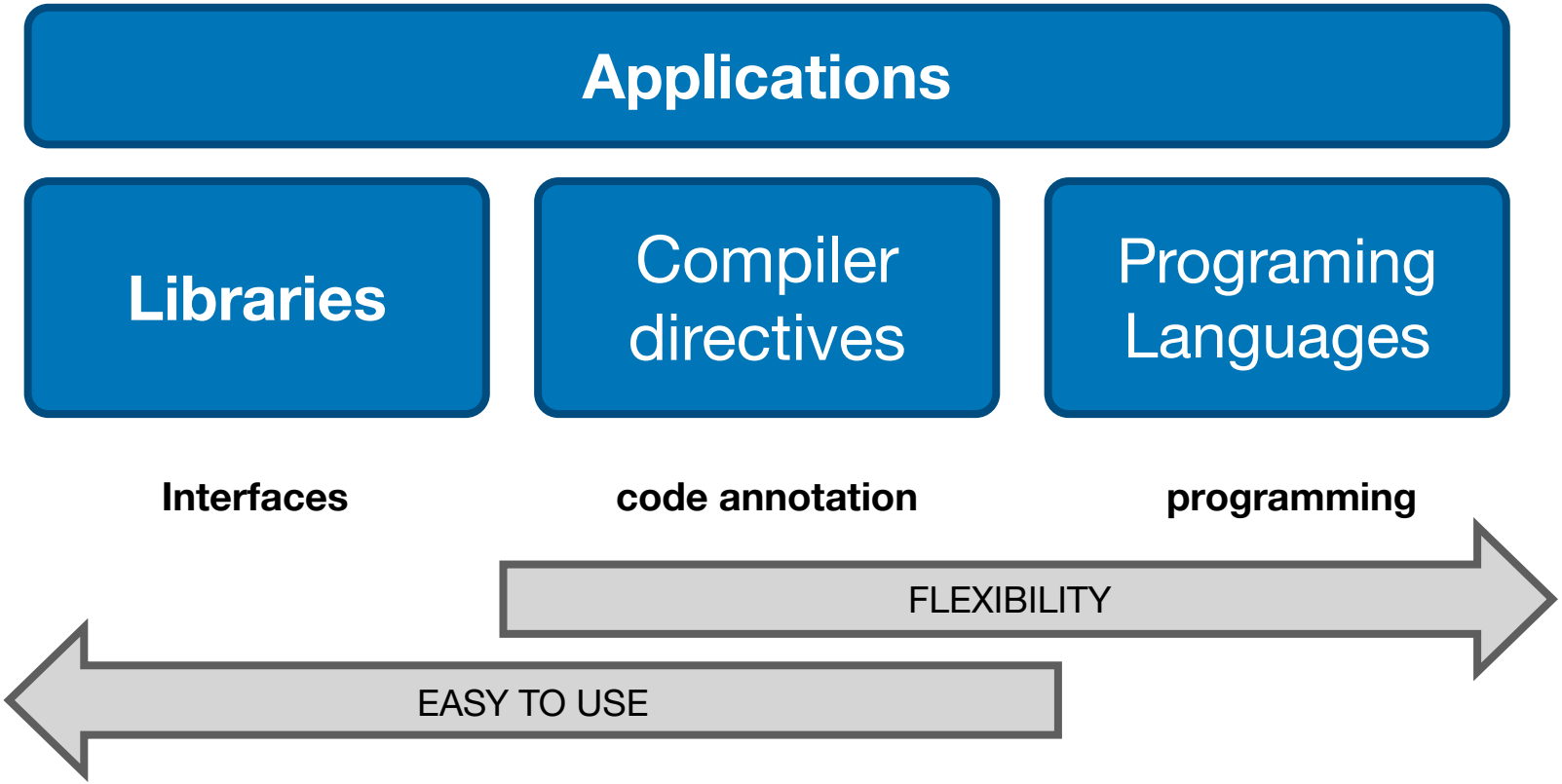
CPU memory ~100GB - 100GB/s

IO Bus

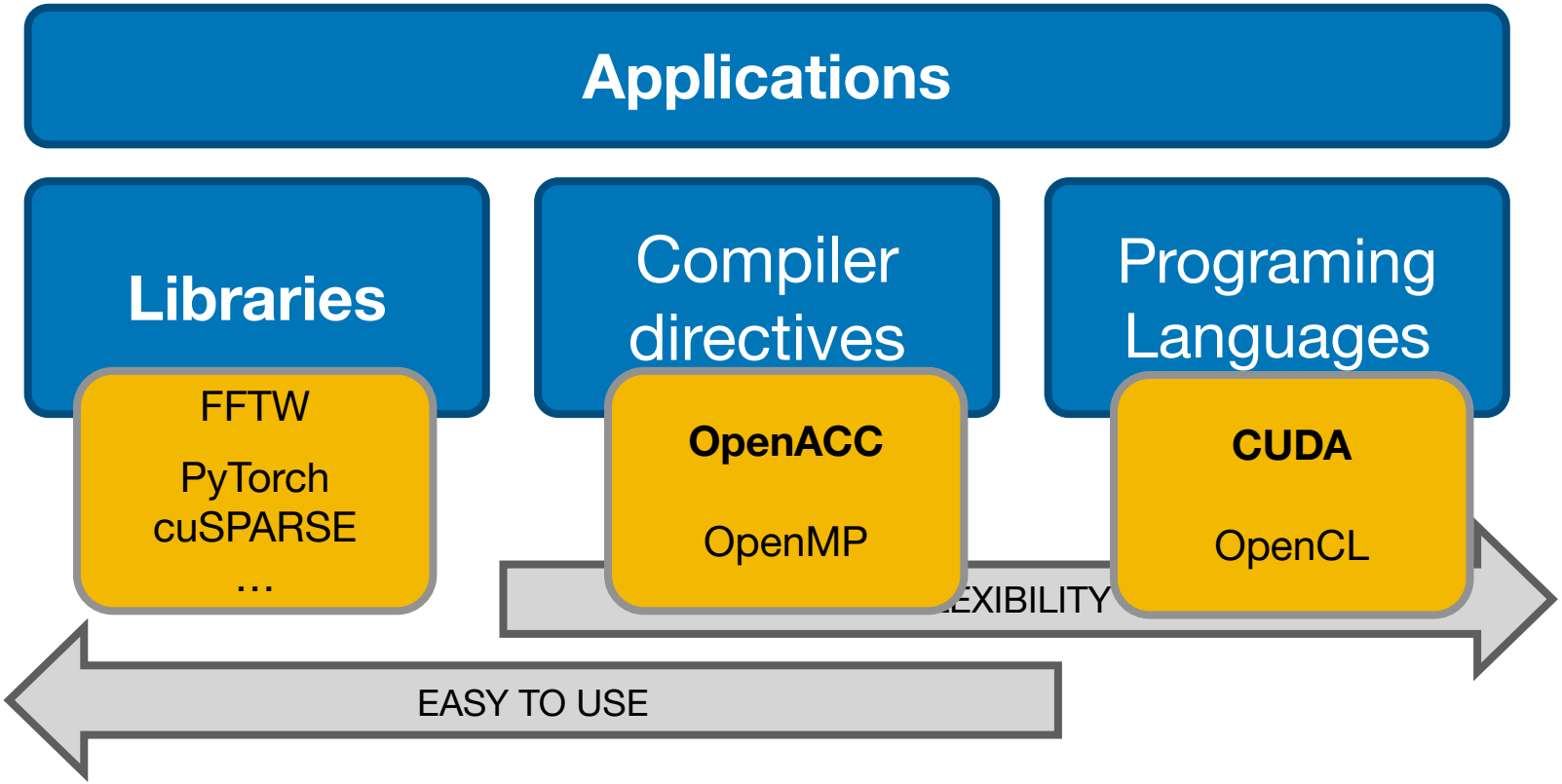
PCI Express 3 x16: 15.75GB/s
PCI Express 6 x16: 242GB/s



Using GPUS


Applications
Libraries
Compiler directives
Programing Languages
Interfaces
code annotation
programming
FLEXIBILITY
EASY TO USE

Using GPUS


Applications
Libraries
Compiler directives
Programing Languages
FFTW
PyTorch
cuSPARSE
…
OpenACC
OpenMP
CUDA
OpenCL
EXIBILITY
EASY TO USE

CUDA architecture outline

Credit: some slides of this sección are based on the online tutorial of openACC.org

Volta GPU

Professional NVIDIA Pascal GV100
architecture with 84 cores using 12nm fab size

HBM2 memory with **900 GB/s bandwidth**
and 5376 ALUs with
peak 15.7 TFLOP/s (SP) **7.8 TFLOP/s (DP)**

Credit: Tim Warburton, ATPESC 2018

Volta GPU

Professional NVIDIA Pascal GV100
architecture with 84 cores using 12nm fab size

HBM2 memory with **900 GB/s bandwidth**
and 5376 ALUs with
peak 15.7 TFLOP/s (SP) **7.8 TFLOP/s (DP)**

Credit: Tim Warburton, ATPESC 2018

Streaming Multiprocessor

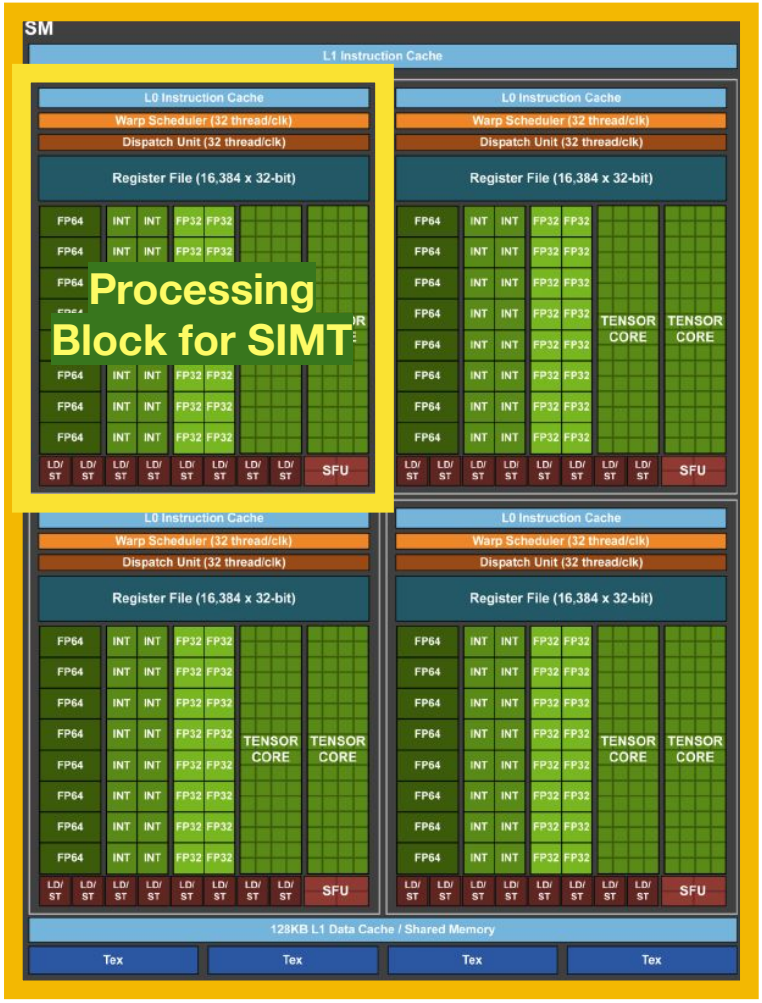
Volta GPU

Professional NVIDIA Pascal GV100
architecture with 84 cores using 12nm fab size

HBM2 memory with **900 GB/s bandwidth**
and 5376 ALUs with
peak 15.7 TFLOP/s (SP) **7.8 TFLOP/s (DP)**

Credit: Tim Warburton, ATPESC 2018

Streaming Multiprocessor



In UPFs cluster

NVidia GTX 1080Ti

- released 2017
- 11GB mem at 480GB/s
- 28 streaming multiprocessors
- 3584 cores
- compute capability up to version 6.1

OpenACC basics

Credit: some slides of this sección are based on the online tutorial of openACC.org

OpenACC

Directives-based parallel programming model designed for **performance portability.**

Born as a branch of OpenMP to offer rapid support for accelerators, but split from OpenMP due to opposing commercial interests of Intel (behind OpenMP) and NVidia (behind OpenACC).

The high-level API consists of:

- Preprocessor (compiler) directives.
- Library routines.
- Environment variables.

It's a standard. Developed by openacc.org.

OpenMP has been extended for GPU programming as well.

Supported Platforms

POWER
Sunway
x86 CPU
AMD GPU
NVIDIA GPU
PEZY-SC

OpenACC Model

Incremental: Start with sequential code and incrementally add annotations to expose parallelism (verifying correctness)

Follow Amdahl's law → parallelize the most time consuming regions first

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

Portable: **Single source code** can be compiled on different architectures. Compiler determines the parallelization! Sequential code is maintained

Not modular: directive-based approaches need to expose parallelism throughout the whole code.

OpenACC syntax

```
#pragma acc <directive> <clauses>
{
    // code
}
```

`#pragma:` gives indications to the compiler, these are not mandatory, the compiler can freely ignore them

`acc:` indicates that is an OpenACC pragma, compilers not supporting OpenACC will ignore the pragma

`directive:` command in OpenACC to alter the code

`clauses:` specifiers or additions to the commands
All these elements together form a so-called **OpenACC construct**

OpenACC directive types

OpenACC distinguishes three directive types:

Compute directives:
mark a block of code that can be accelerated by exploiting inherent parallelism and distribute work to multiple threads

Data management directives:
specify data lifetimes to avoid unnecessary transfers between host and device

Synchronization directives:
to explicitly wait for the competition of concurrent tasks

Credit: OpenACC for Programmers: Concepts and Strategies.

OpenACC **`loop`**

For GPUs, we will only perform loop parallelism (no tasks). The most used compute directive is **`loop`** directive.

The induction variable **`i`** is privatized. Each thread has its own private copy.

OpenACC uses a hierarchical organization of threads (gangs, workers, vector). We will see that in the next lectures.

```
#pragma acc parallel loop
for(i=0; i < size; i++) {
    for(j=0; j < size; j++) {
        // code here ...
    }
}
```

A single `loop` directive can parallelize nested loops using this hierarchy.

OpenACC **parallel**

There's a also a **parallel** directive similar to OpenMP.

loop can be used **within a parallel region,** or merged with **parallel**.

```
#pragma acc parallel
{
    // thread gangs are created
    // and all gangs execute code here
    int x = foo();

    #pragma acc loop
    for(i=0; i < size; i++) {
        // for distribute work to threads
    }
}

#pragma acc parallel loop
for(i=0; i < size; i++) {
    // do something
}
```

Data Scope

Iterators are by default `private`

scalars are by default `firstprivate`

Any variable (scalar or not) declared within the loop will be made `private`

Arrays are by default `shared`

Parallelize loops

Loop parallelism divides the iterations among the set of threads.

- There can be **no dependency** among iterations.
- Loops must follow a form that allows the compiler to compute the number of iterations at compile time.
- Loops with calls to external functions cannot be parallelized (we will see how this can be done).
- Valid *induction variables* are: integers, pointers and iterators. These are automatically set as **`private`**.

```
while (err < tolerance) {
    x = update(x);
    err = compute_error(x)
}

for (int i=0; i<N; i++) {
  int a = my_function(i);
  if (a < 0) {
      break;
  }
}
```

Reduction

```
#pragma acc parallel loop reduction(+:total)
for(i=0; i < N; i++) {
    total += arr[i];
}
```

`reduction(operation: variable)`, where `operation` can be:

- arithmetic operations **`+`**, **`*`**, **`min`**, **`max`**
- logical `||`, **`&&`**
- bitwise operations `|`, **`&`**, **`^`**

GPU implementation of a reduction operation is different than the multicore version.

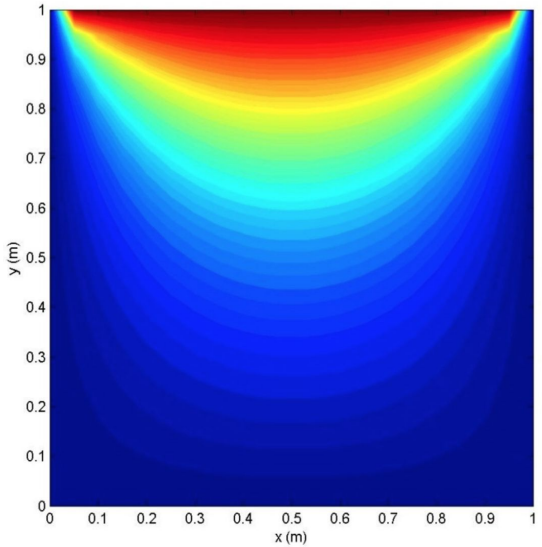
Example: Heat conductivity problem

- Heat transfer in a metal can be modeled by the Poisson's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Omega = [-1,1] \times [-1,1]$$

- We apply consistent heat at the top of the plate and keep the temperature at the rest of the boundaries equal to 0

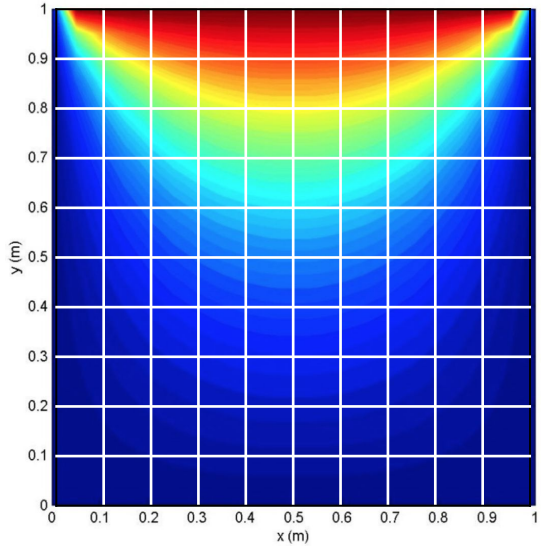
- We will simulate the heat distribution across the plate

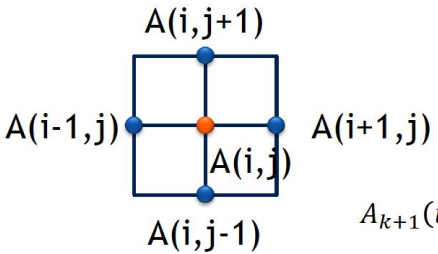


Jacobi algorithm

The Jacobi algorithm is a classical relaxation method for the solution of linear systems.

In this case the temperature at each node of the grid is computed as the average of its neighbors until convergence is reached…

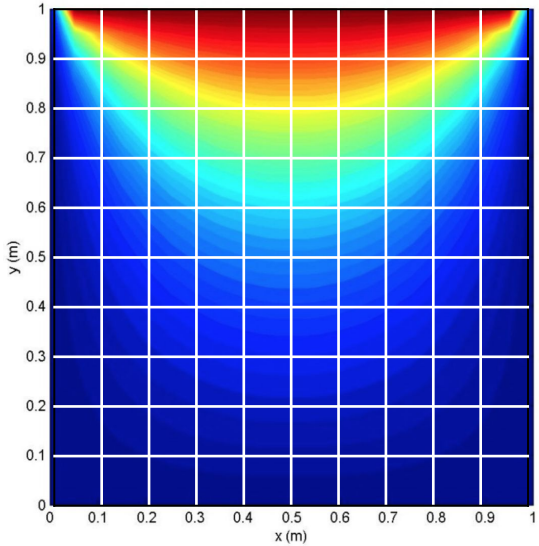




$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

OpenACC cycle

1. Identify parallelism - which parts of the could should and could be parallelized?
2. Express parallelism - indications to the compiler
3. Express data movements
4. Optimize loops
5. Go back to 1!



Main code (heat.c)

```
  ...

printf("Jacobi algorithm, %d x %d mesh\n", nx, ny);

double st = omp_get_wtime();
while ( error > tol && iter < iter_max )
{
   error = iterJacobi(T, Tnew, nx, ny);
   swap(T, Tnew, nx, ny);

   if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

   iter++;

}
double runtime = omp_get_wtime() - st;

saveResult(T);
printf(" total: %f s\n", runtime);



  ...
```

Functions (jacobi.c)

```
#define ind(i, j, nx) (((j)*(nx)) + (i))

double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;

   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(i, j-1, nx)] + T[ind(i, j+1, nx)]);
         error = fmax(error,fabs(Tnew[ind(i, j, nx)] - T[ind(i, j , nx)]));
      }
   }
   return error;
}

void swap(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   for( int j = 1; j < ny-1; j++)
      for( int i = 1; i < nx-1; i++)
         T[ind(i, j, nx)] = Tnew[ind(i, j, nx)];
}
```

Sequential execution

- Result sequential execution

```
[bsc21665@p9login1 seq]$ ./heat 4096 4096 500
Jacobi algorithm, 4096 x 4096 mesh
  100, 0.002421
  200, 0.001210
  300, 0.000806
  400, 0.000605
  500, 0.000484
 total: 24.081053 s
```

- While we optimize the code we will check that the numerical result does not change…

Exposing parallelism in functions…

```
#define ind(i, j, nx) (((j)*(nx)) + (i))

double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;
   #pragma acc parallel loop reduction(max:error)
   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(i, j-1, nx)] + T[ind(i, j+1, nx)]);
         error = fmax(error,fabs(Tnew[ind(i, j, nx)] - T[ind(i, j , nx)]));
      }
   }
   return error;
}

void swap(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   #pragma acc parallel loop
   for( int j = 1; j < ny-1; j++)
      for( int i = 1; i < nx-1; i++)
         T[ind(i, j, nx)] = Tnew[ind(i, j, nx)];
}
```

Compilation

We will use the `nvc` compiler (not to be confused with `nvcc`). New versions of gcc also support OpenACC.

`nvc` is a C11 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs. `nvc` supports GPU programming with OpenACC, and supports multicore CPU programming with OpenACC and OpenMP.

`nvc` is developed by NVIDIA, as part of the NVIDIA HPC SDK (previously it was known as the PGI compiler `pgcc`).

In the cluster, load NVIDIA HPC SDK module:

```
module load NVHPC
```

Compilation

Compilation for sequential execution:

```
nvc -fast -Minfo=all heat.c jacobi.c -mp -o heat
```

Compilation for multicore CPU:

```
nvc -fast -acc=multicore -Minfo=all heat.c jacobi.c -mp -o heat
```

Compilation for GPU:

```
nvc -acc=gpu -gpu=managed -Minfo=all heat.c jacobi.c -mp -o heat
```

`-fast` : optimize CPU code (similar to `-O2` in `gcc`)
`-acc` : enable OpenACC directives.
We can specify `host` (single core), `multicore` (multiple cores), and `gpu`
`-gpu` : options for GPU compilation
`-mp` : enable OpenMP
`-Minfo`: provide information about compilation → **very useful**

`nvc` compiler flags

More info about compiler options here.

`-fast` : optimize CPU code (similar to `-O2` in `gcc`)
`-mp` : enable OpenMP

`-acc` : enable OpenACC directives.
- `-acc=host` → single core (sequential code)
- `-acc=multicore` → multiple cores
- `-acc=gpu`

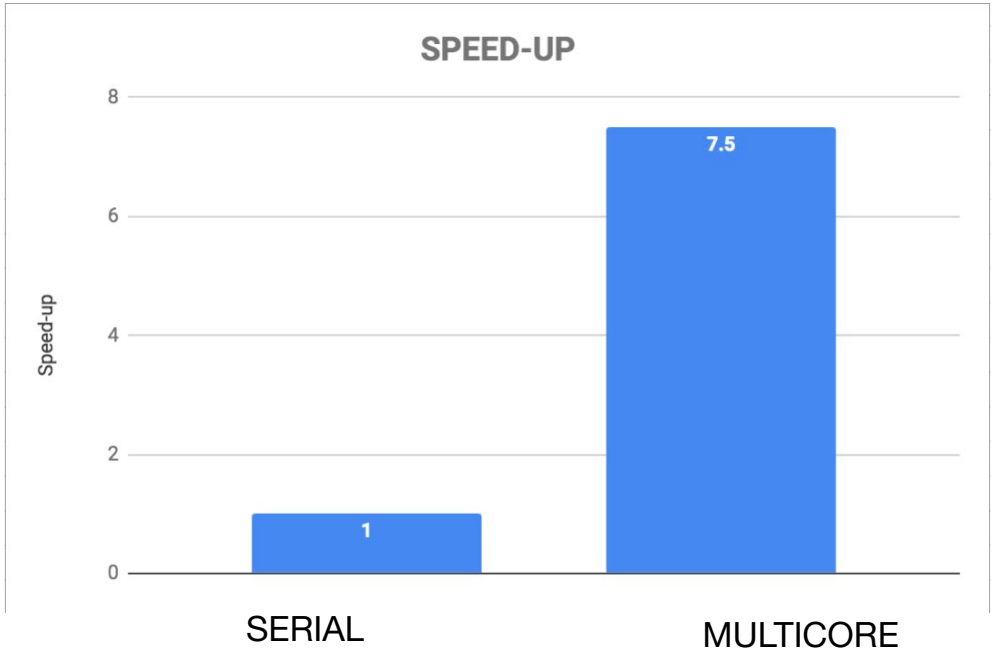
`-gpu` : options for GPU compilation
- `-gpu=managed` → manage memory for us (more on this later)
- `-gpu=cc60,cc70` → compute capability 6.0 & 7.0

`-Minfo`: provide information about compilation → **very useful**
- `-Minfo=accel` → info about code accelerated with OpenACC directives
- `-Minfo=opt` → info about code optimization
- `-Minfo=all` → all info

Compilation output multicore CPU

```
heat.c:
jacobi.c:
iterJacobi:
     10, Generating Multicore code
         11, #pragma acc loop gang
     11, Accelerator restriction: size of the GPU copy of Tnew,T is unknown
         Generating reduction(max:error)
     13, Accelerator restriction: size of the GPU copy of T is unknown
         Loop is parallelizable
swap:
     26, Generating Multicore code
         27, #pragma acc loop gang
     27, Accelerator restriction: size of the GPU copy of Tnew,T is unknown
     28, Loop is parallelizable
```

Result with POWER9 CPU

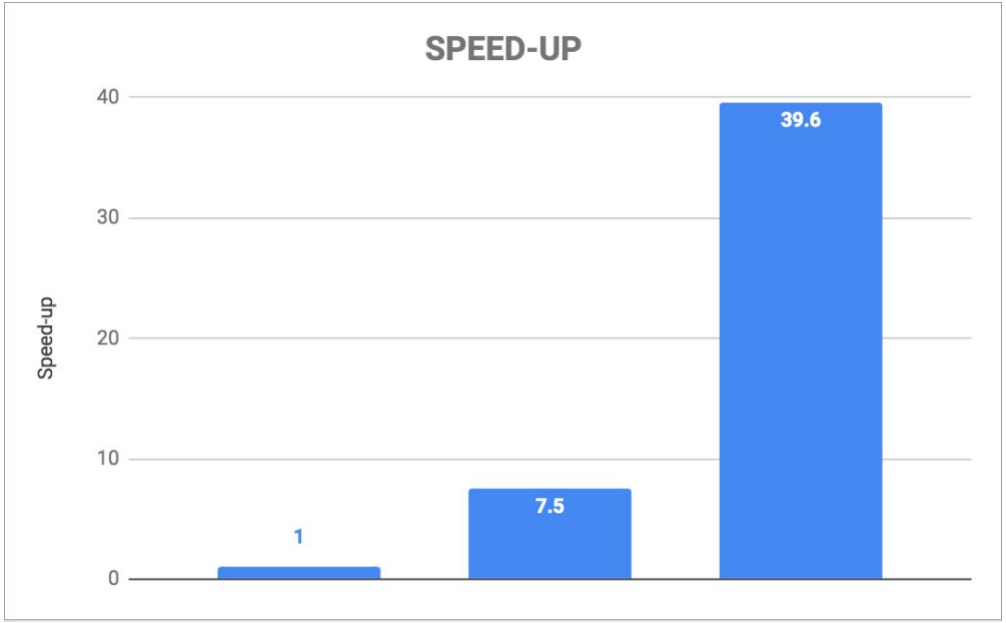


500 Jacobi iterations mesh 4096 x 4096

Compilation output GPU

```
heat.c:
jacobi.c:
iterJacobi:
     10, Generating Tesla code
         11, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         13, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating implicit copyin(T[:])
         Generating implicit copy(error)
         Generating implicit copyout(Tnew[:])
     13, Loop is parallelizable
swap:
     26, Generating Tesla code
         27, #pragma acc loop gang /* blockIdx.x */
         28, #pragma acc loop vector(128) /* threadIdx.x */
     26, Generating implicit copyout(T[:])
         Generating implicit copyin(Tnew[:])
     28, Loop is parallelizable
```

Result with NVIDIA Volta GPU



500 Jacobi iterations mesh 4096 x 4096

Few questions…

Exposing parallelism…

…

```
printf("Jacobi algorithm, %d x %d mesh\n", nx, ny);

double st = omp_get_wtime();
while ( error > tol && iter < iter_max )
{
   error = iterJacobi(T, Tnew, nx, ny);
   swap(T, Tnew, nx, ny);

   if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

   iter++;

}
double runtime = omp_get_wtime() - st;

saveResult(T);
printf(" total: %f s\n", runtime);
```

Can this loop be parallelized?

…

Exposing parallelism…

…

```
printf("Jacobi algorithm, %d x %d mesh\n", nx, ny);

double st = omp_get_wtime();
while ( error > tol && iter < iter_max )
{
   error = iterJacobi(T, Tnew, nx, ny);
   swap(T, Tnew, nx, ny);

   if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

   iter++;

}
double runtime = omp_get_wtime() - st;

saveResult(T);
printf(" total: %f s\n", runtime);
```

Can this loop be parallelized?

NO… each iteration depends on the previous one

Even if we try to parallelize it with OpenACC, the compiler will detect the dependency and will skip parallelization.

…

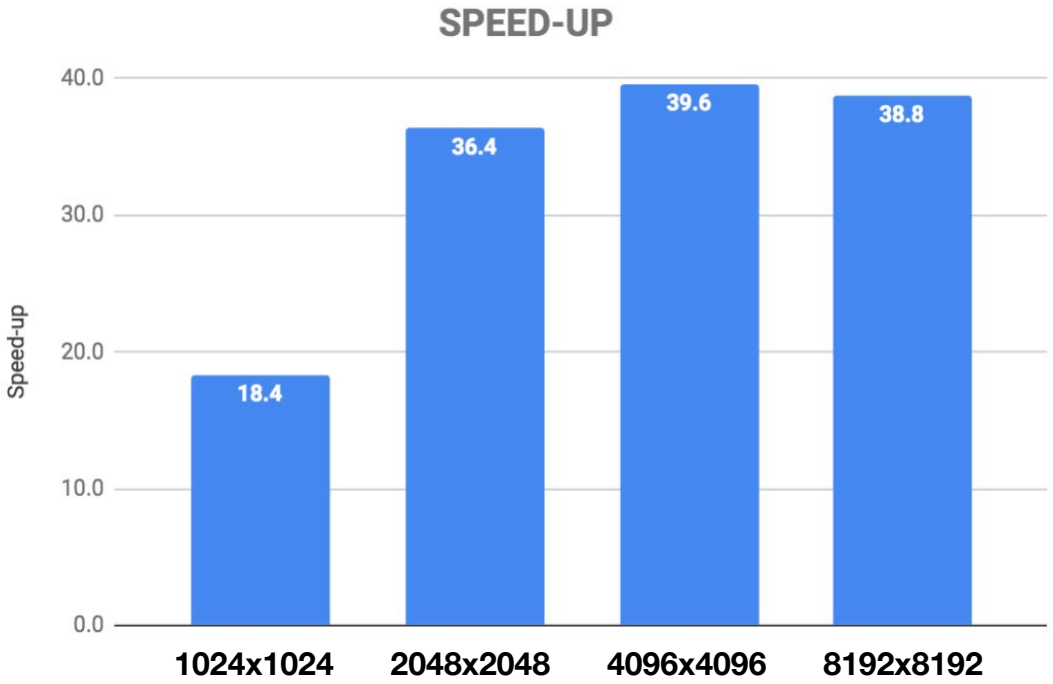
Does the speedup GPU vs CPU depend on the mesh size?

Sequential vs GPU

GPU requires **occupancy** to achieve its peak performance.

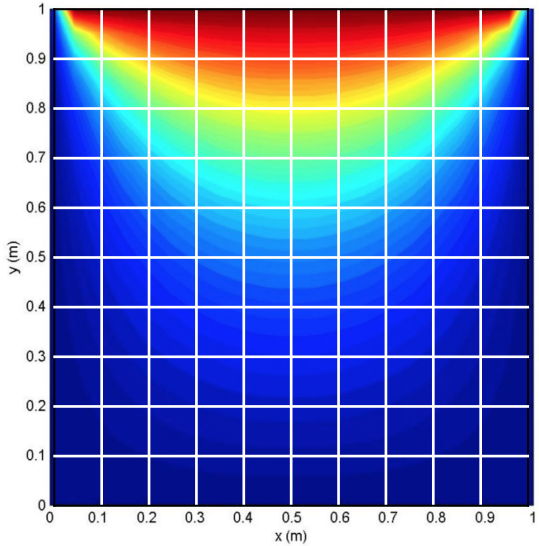
It's advisable to have more threads running than the number of GPU cores.

On the other hand, CPU memory hierarchy optimizes the performance for small problem sizes.



Is the loops order (first j loop then i loop) important in terms of performance?

```
#define ind(i, j, nx) (((j)*(nx)) + (i))

void swap(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
    #pragma acc parallel loop
    for( int j = 1; j < ny-1; j++)
      for( int i = 1; i < nx-1; i++ )
         T[ind(i, j, nx)] = Tnew[ind(i, j, nx)];
}
```



Performance and loops ordering

	SEQUENTIAL	MULTICORE	v100 GPU
ORDER j,i	3.8s	0.6s	0.26s
ORDER i,j	77.6s	3.7s	0.41s
slowdown	**20.3x**	**6.4x**	**1.58x**

It is very important in CPU code because of cache misses.

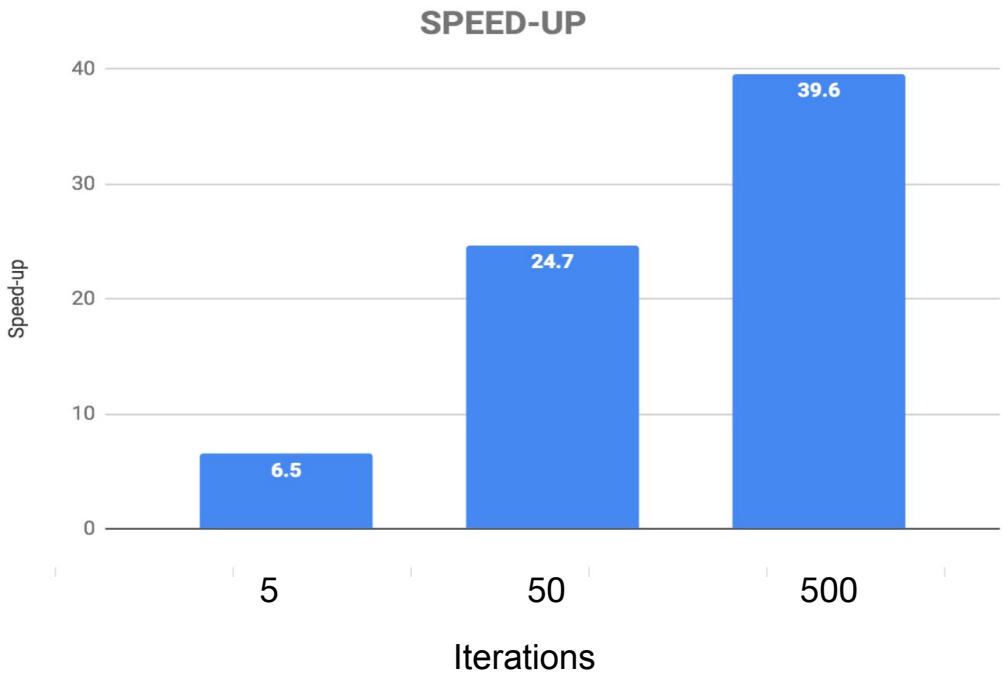
It is somewhat important on GPU, but much less so, because GPUs are more robust to latency.

In any, case it is a suboptimal code.

500 swap iterations mesh 4096 x 4096

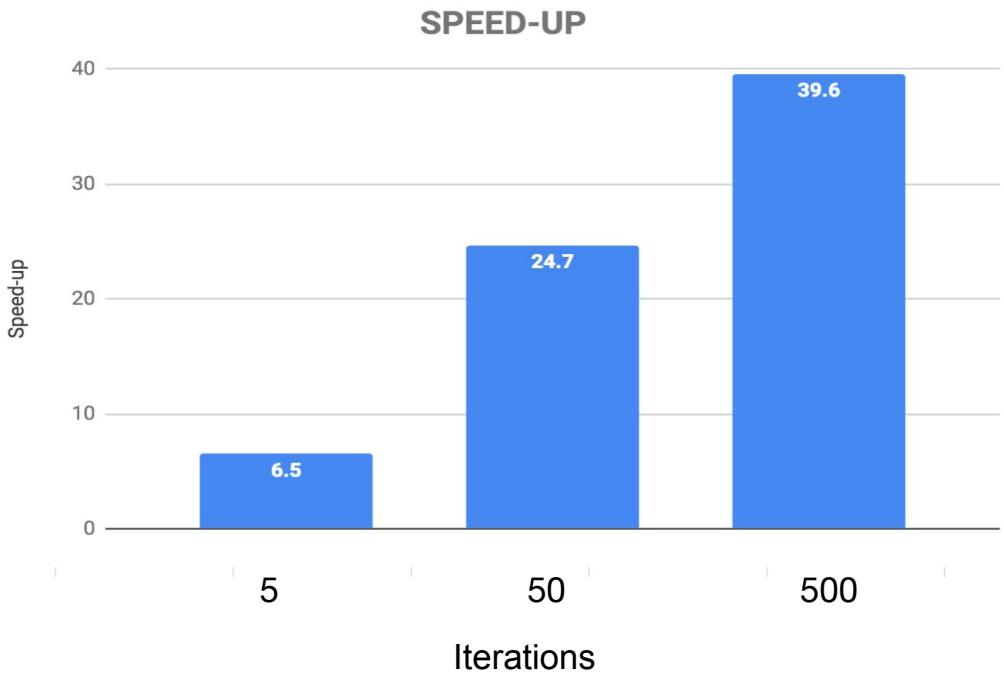
Does the performance ratio depend on the number of Jacobi iterations?

Performance and Jacobi iterations

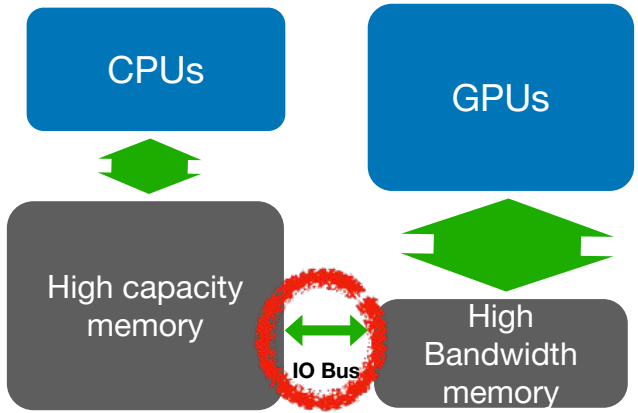


The mesh size is 4096 x 4096.

Performance and Jacobi iterations



The mesh size is 4096 x 4096.



T and Tnew are moved only once to the GPU, the overhead of the data transfer depends thus on the iterations

OpenACC Data management

Credit: some slides of this sección are based on the online tutorial of openACC.org

GPU data management

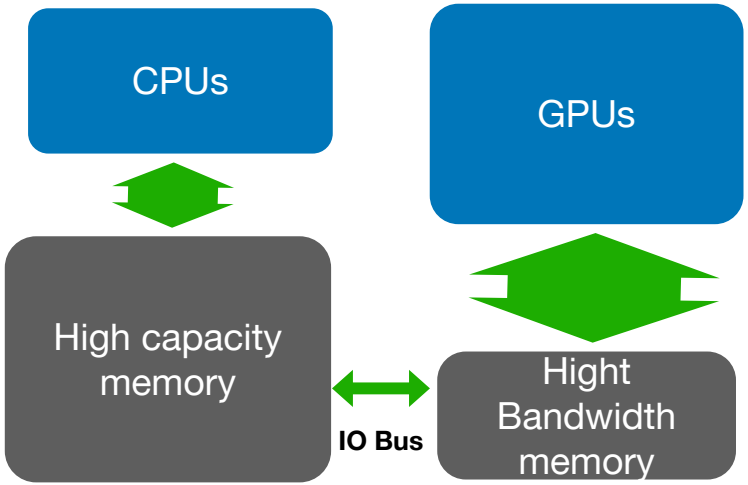
CPU memory is **larger**

GPU memory **has more bandwidth**

CPU and GPU memories are separated by a I/O bus (PCI-e)

Processing flow:

1. **Copy input** from CPU RAM to the GPU
2. **Execute** on the GPU
3. **Copy results** from GPU to CPU RAM



CUDA unified memory

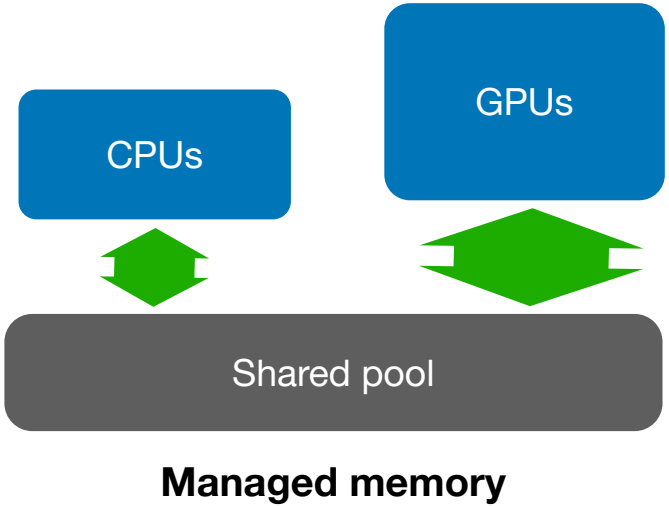
```
nvc -acc=gpu -gpu=managed -Minfo=all heat.c jacobi.c -mp -o heat
```

Compiler handles transfers between CPU and GPU memories.

This simplifies development. However, data transfers handled by the programmer can outperform the managed memory.

Managed memory **does not allow** asynchronous data transfers and allocations/deallocations take longer.

Currently only supported by `nvc` compiler.



Compilation on GPU w/o managed flag

```
heat.c:
jacobi.c:
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages
): Could not find allocated-variable index for symbol - T (jacobi.c: 10)
iterJacobi:
     10, Generating Tesla code
         11, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         13, #pragma acc loop vector(128) /* threadIdx.x */
     11, Accelerator restriction: size of the GPU copy of Tnew,T is unknown
     13, Accelerator restriction: size of the GPU copy of T is unknown
         Loop is parallelizable
PGC-F-0704-Compilation aborted due to previous errors. (jacobi.c)
PGC/x86-64 Linux 19.4-0: compilation aborted
```

OpenACC Data Clauses

If we don't use managed memory, we need to manage data ourselves using data clauses.

`copy(list)`: Allocates memory on GPU. Copies data from host to GPU when entering the data region, and from GPU to host when exiting the data region. Allocated memory is deallocated.

`copyin(list)`: Allocates memory on GPU. Copies data from host to GPU when entering the data region. On exit deallocate memory on the device if it was allocated on entry.

`copyout(list)`: Allocates memory on GPU. Copies data from GPU to host when exiting the data region. On exit deallocate memory on the device if it was allocated on entry.

`create(list)`: On entry: allocates memory on GPU (unless allocated already). If memory was allocated, it is deallocated on exit.

`present(list)`: assume that memory is allocated and that data is present on the device.

Data Clauses Syntaxis

```
#pragma acc parallel loop copyin(x[0:N], A[0:N][0:N]) copyout(y[0:N])
```

Additionally to the appropriate data closure, **the array shape needs to be provided to the compiler.**

Copy indices between `start` and `start+length`:
`copy(array[start:length])`

Multi-dimensional array shaping: `copy(array[0:N][0:M])`

Partial array copy: `copy(array[N/4,N/2])`

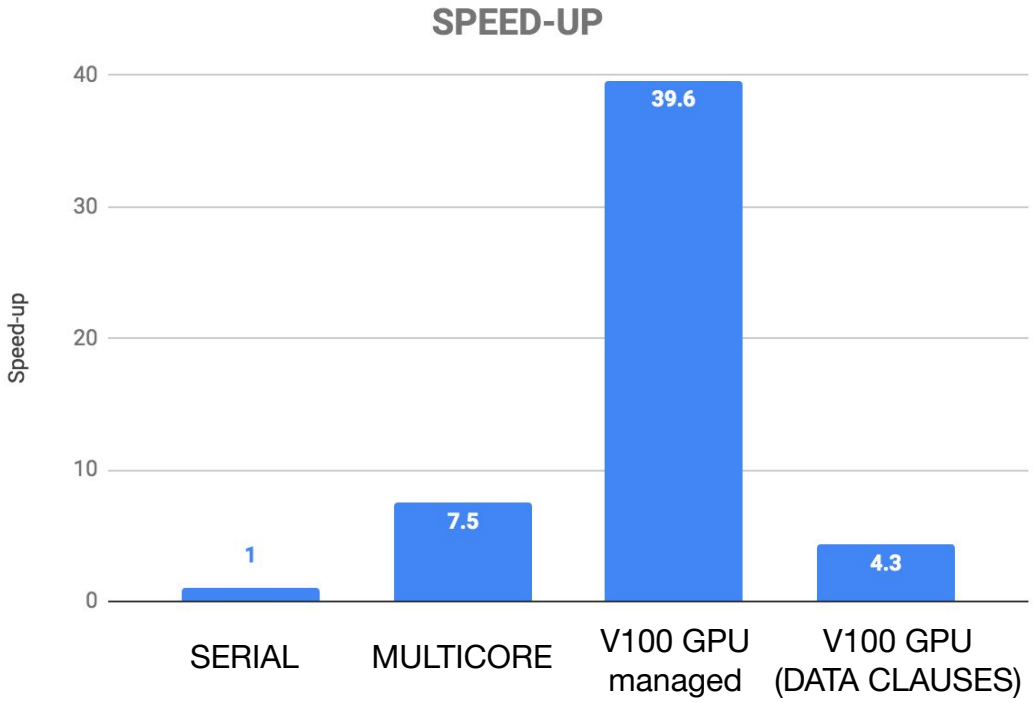
Code annotations for data management

```
double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;

   #pragma acc parallel loop reduction(max:error) copyin(T[0:nx*ny]) copyout(Tnew[0:nx*ny])
   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(i, j-1, nx)] + T[ind(i, j+1, nx)]);
         error = fmax(error,fabs(Tnew[ind(i,j,nx)] - T[ind(i,j,nx)]));
      }
   }
   return error;
}

void swap(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   #pragma acc parallel loop copy(T[0:nx*ny]) copyin(Tnew[0:nx*ny])
   for( int j = 1; j < ny-1; j++)
      for( int i = 1; i < nx-1; i++ )
         T[ind(i, j, nx)] = Tnew[ind(i, j, nx)];
}
```

Compilation on GPU w/o managed flag

```
heat.c:
jacobi.c:
iterJacobi:
     10, Generating copyin(T[:ny*nx])
         Generating Tesla code
         11, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         13, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating implicit copy(error)
         Generating copyout(Tnew[:ny*nx])
     13, Loop is parallelizable
swap:
     26, Generating copy(T[:ny*nx])
         Generating copyin(Tnew[:ny*nx])
         Generating Tesla code
         27, #pragma acc loop gang /* blockIdx.x */
         28, #pragma acc loop vector(128) /* threadIdx.x */
     28, Loop is parallelizable
```

Result with NVIDIA Volta GPU



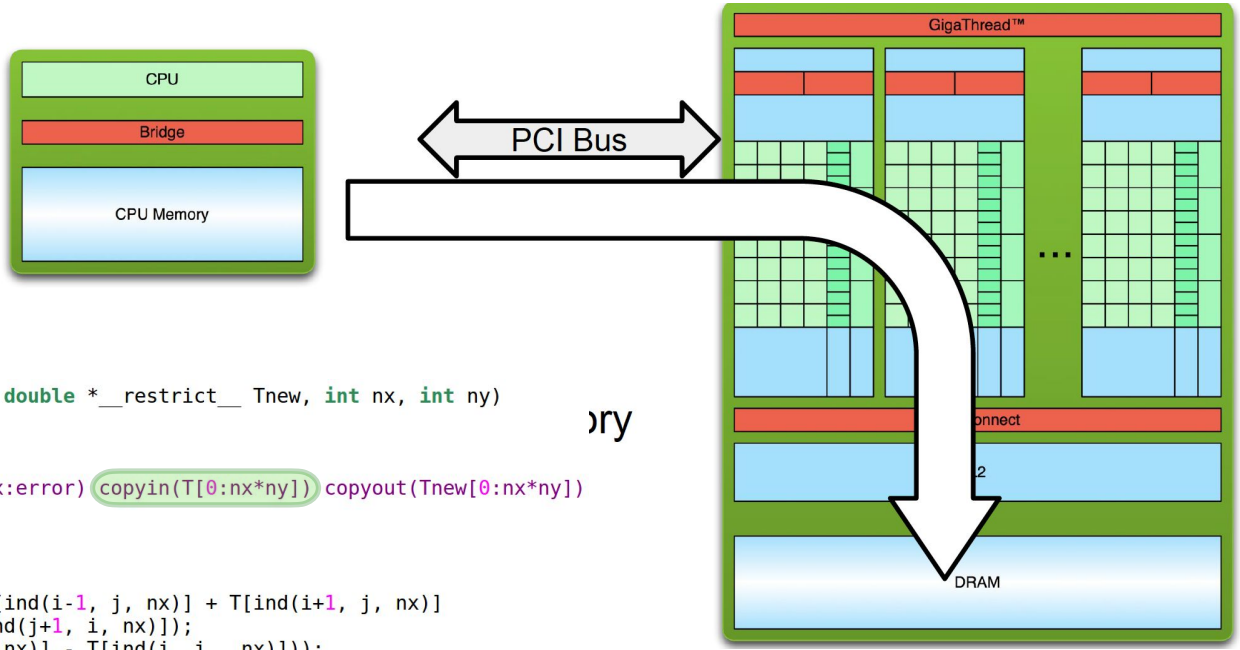
500 Jacobi itarations mesh 4096 x 4096

What is wrong?

Data management and processing flow

Simple processing flow:

1) Copy input data from the CPU to the GPU memory



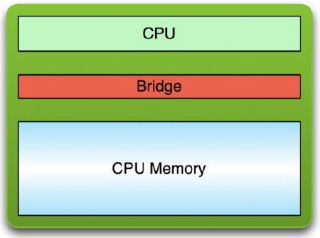

```
double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;

   #pragma acc parallel loop reduction(max:error) copyin(T[0:nx*ny]) copyout(Tnew[0:nx*ny])
   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(j-1, i, nx)] + T[ind(j+1, i, nx)]);
         error = fmax(fabs(Tnew[ind(i, j, nx)] - T[ind(i, j , nx)]));
      }
   }
   return error;
}
```

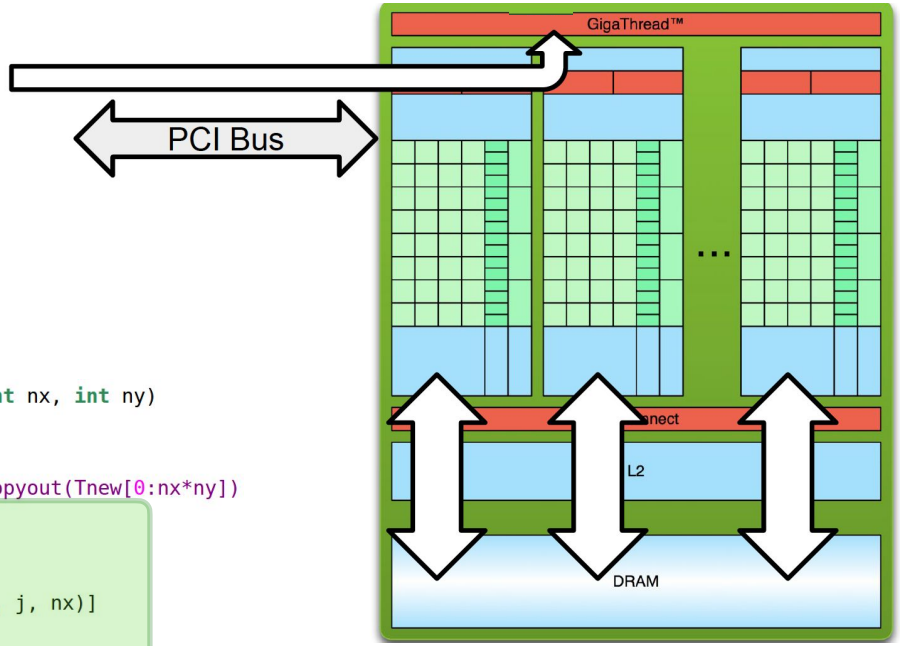
Data management and processing flow

Simple processing flow:

1) Copy input data from the CPU to the GPU memory

2) Load GPU program and execute






```
double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;

   #pragma acc parallel loop reduction(max:error) copyin(T[0:nx*ny]) copyout(Tnew[0:nx*ny])
   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(j-1, i, nx)] + T[ind(j+1, i, nx)]);
         error = fmax(fabs(Tnew[ind(i, j, nx)] - T[ind(i, j , nx)]));
      }
   }
   return error;
}
```

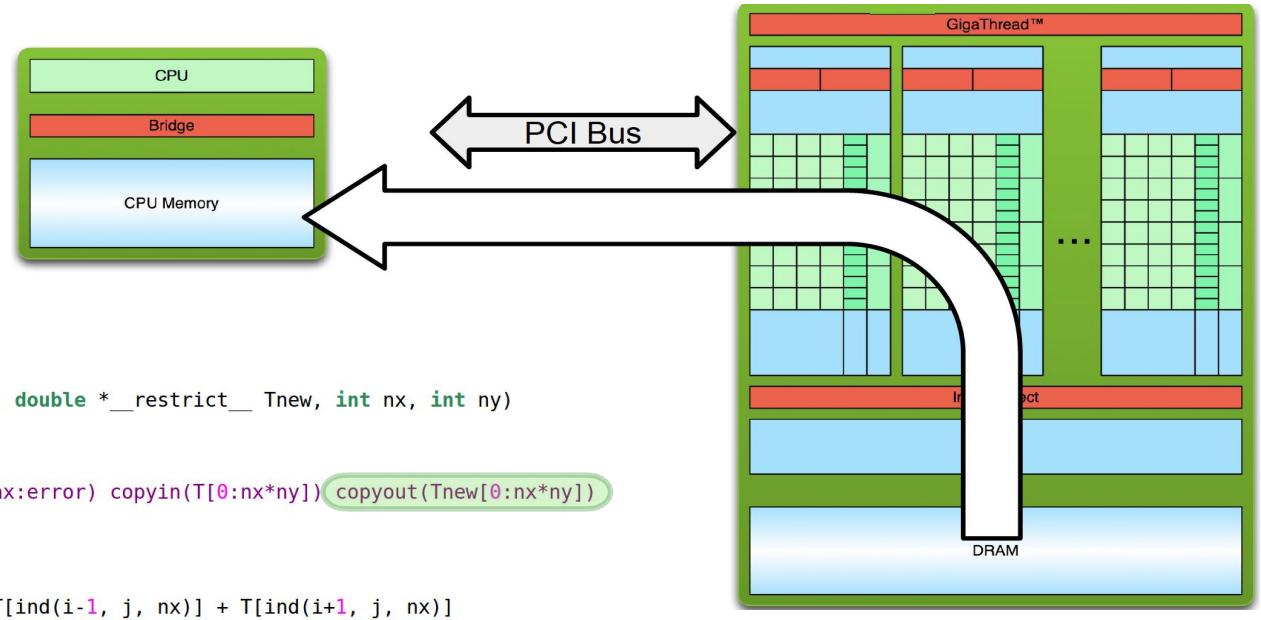
Data management and processing flow

Simple processing flow:

1) Copy input data from the CPU to the GPU memory

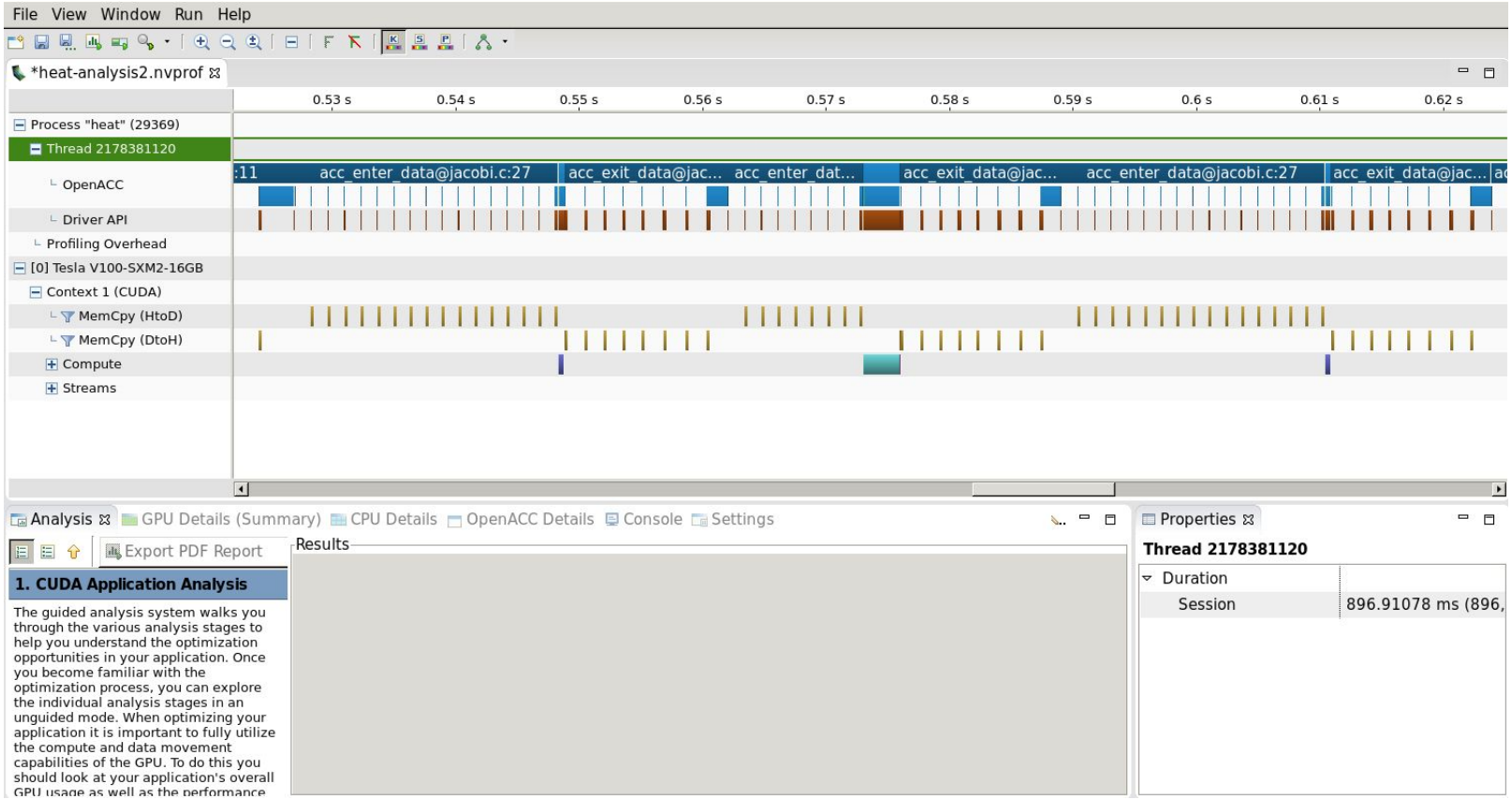
2) Load GPU program and execute

3) **Copy results from GPU memory to CPU memory**

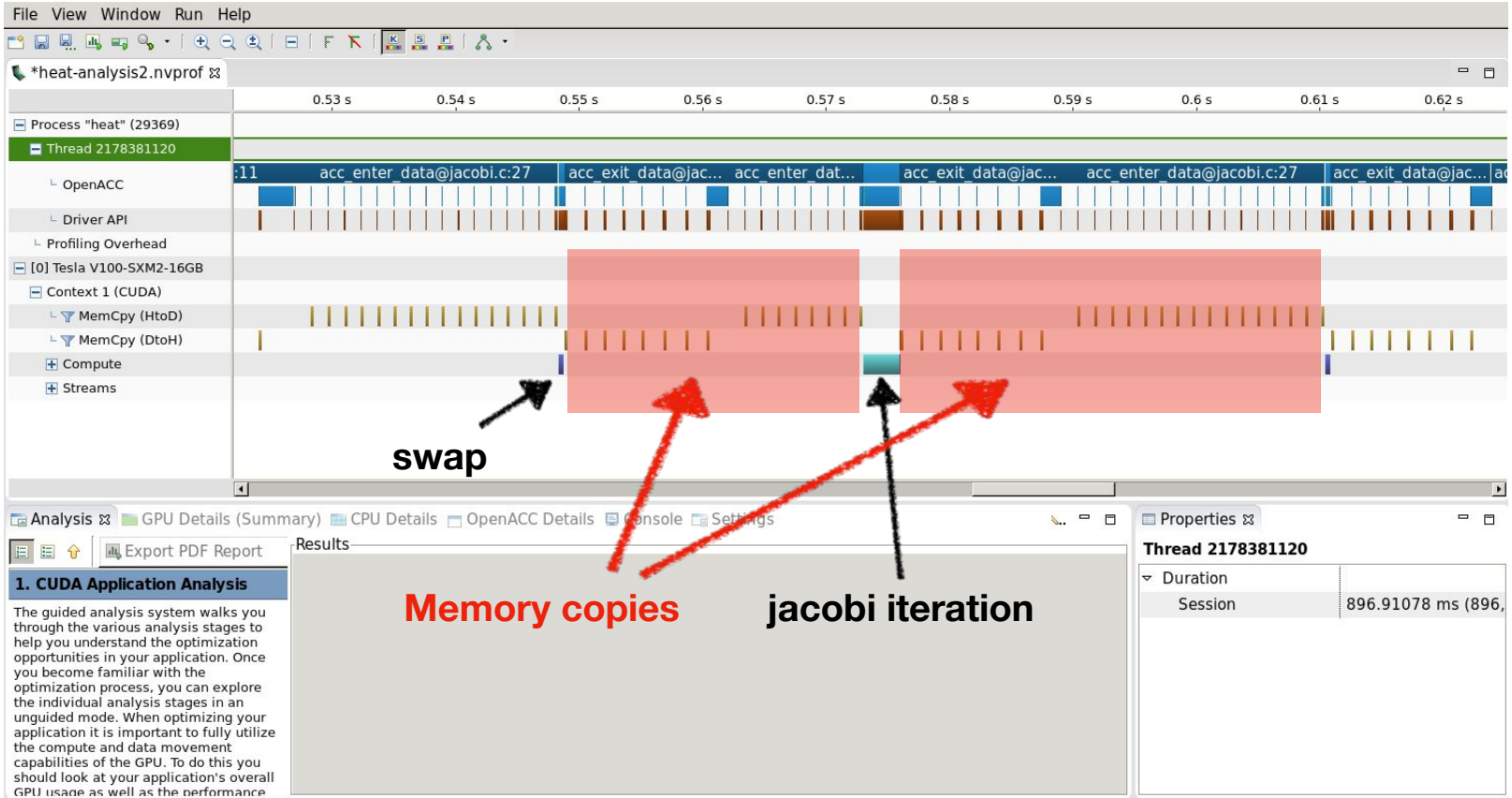



```
double iterJacobi(double *__restrict__ T, double *__restrict__ Tnew, int nx, int ny)
{
   double error = 0.0;

   #pragma acc parallel loop reduction(max:error) copyin(T[0:nx*ny]) copyout(Tnew[0:nx*ny])
   for( int j = 1; j < ny-1; ++j )
   {
      for( int i = 1; i < nx-1; ++i)
      {
         Tnew[ind(i, j, nx)] = 0.25 * ( T[ind(i-1, j, nx)] + T[ind(i+1, j, nx)]
               + T[ind(j-1, i, nx)] + T[ind(j+1, i, nx)]);
         error = fmax(fabs(Tnew[ind(i, j, nx)] - T[ind(i, j , nx)]));
      }
   }
   return error;
}
```

Let’s see the profiler



Let’s see the profiler



OpenACC data regions

Two concepts are important related to data objects:

Data lifetime: starts when an object is first made available to a device, and ends when the object is no longer available on the device. Typically, data on the device has the same lifetime as the OpenACC construct it was declared in.

We have two options for managing data lifetime. **Structured** and **unstructured** data regions.

Credit: OpenACC for Programmers: Concepts and Strategies.

Structured data region

The **`data`** directive defines a lifetime for data on the device beyond individual loops.

A **structured data region** defines in a single OpenACC construct the data lifetime.

Within the region data is accessible by the device.

Data clauses express a certain data handling.

```
#pragma acc data clauses
{

   <code>

}
```

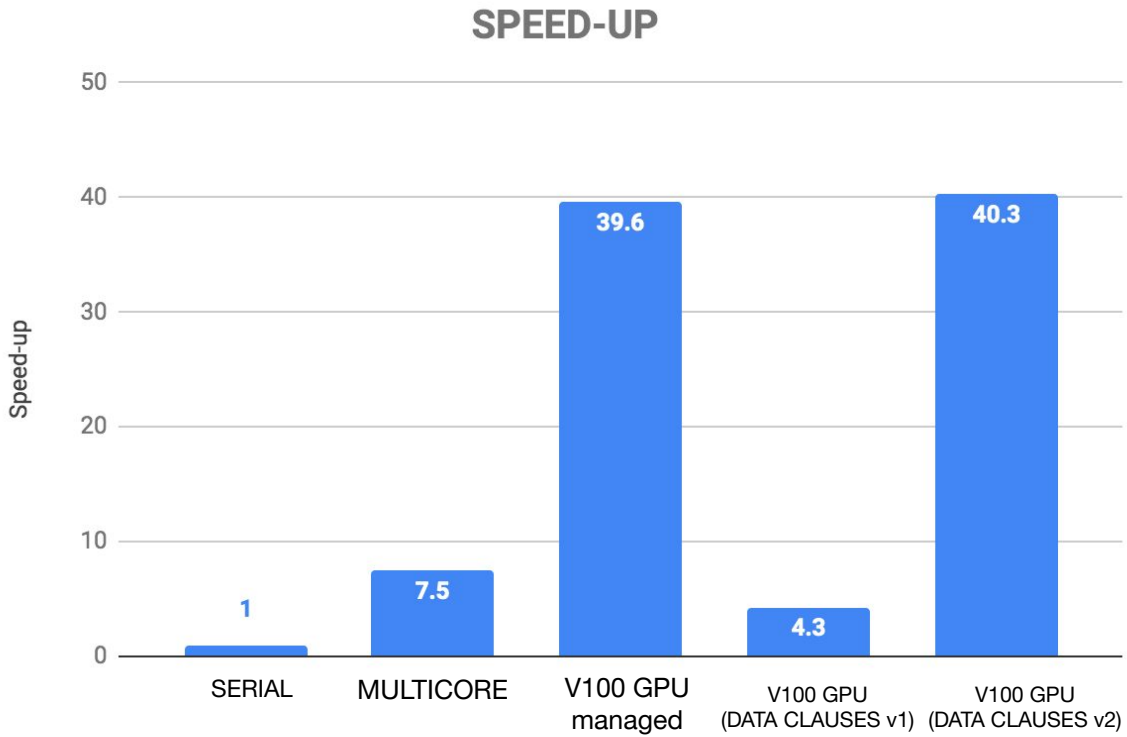
Code annotations for data management

```
double st = omp get wtime();
#pragma acc data copy(T[0:nx*ny]) create(Tnew[0:nx*ny])
while ( error > tol && iter < iter max )
{
   error = iterJacobi(T, Tnew, nx, ny);
   swap(T, Tnew, nx, ny);

   if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

   iter++;

}
double runtime = omp get wtime() - st;

printf(" total: %f s\n", runtime);
```

Compilation on GPU w/o managed flag

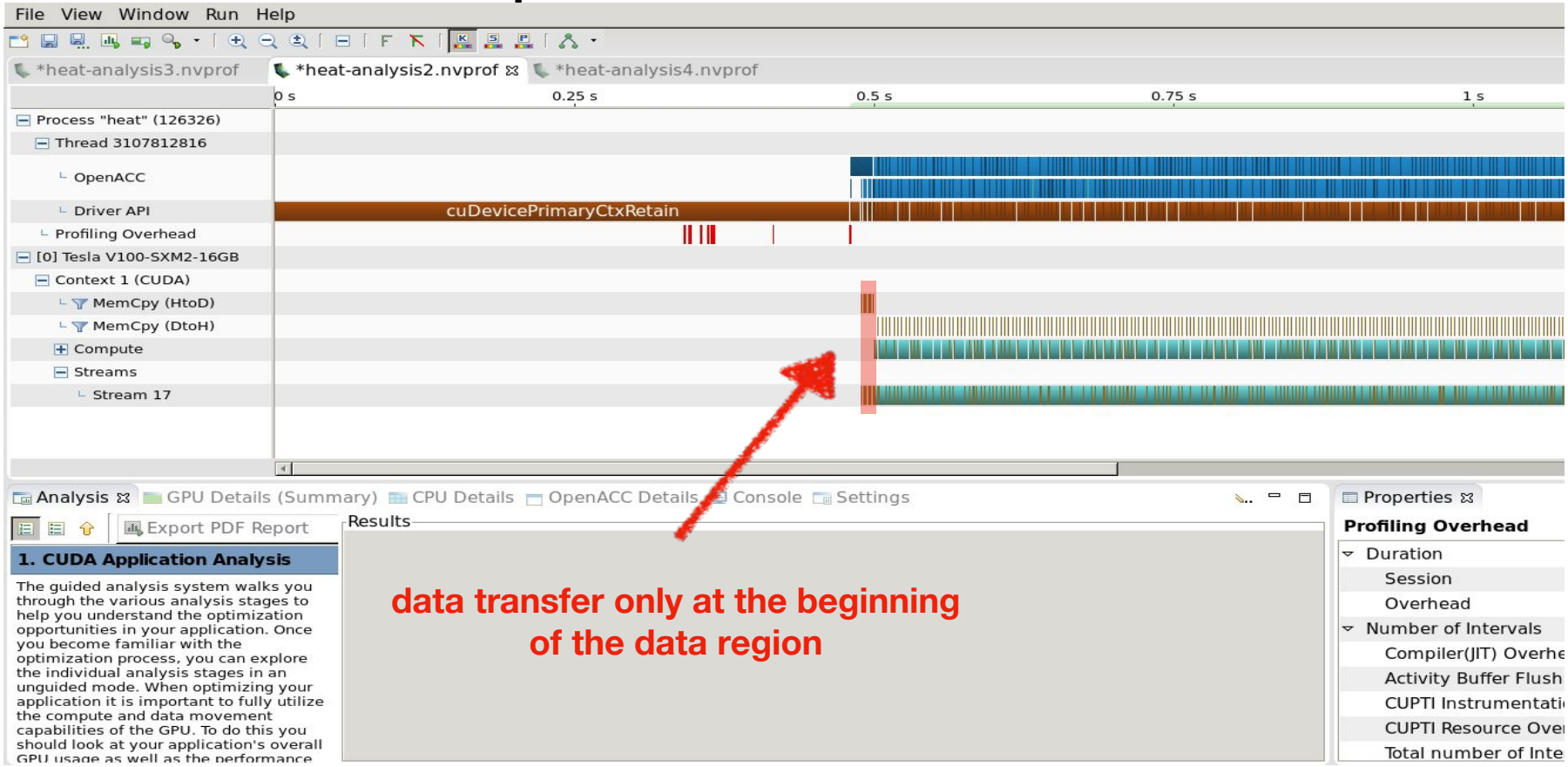
```
rborrell@node029:~/codi/paral$ pgcc -fast -ta=tesla -Minfo=accel    heat.c jacobi.c -mp -o heat

heat.c:
main:
     25, Generating create(Tnew[:ny*nx])
         Generating copyin(T[:ny*nx])
jacobi.c:
iterJacobi:
     10, Generating copyin(T[:ny*nx])
         Generating Tesla code
         11, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         13, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating implicit copy(error)
         Generating copyout(Tnew[:ny*nx])
     13, Loop is parallelizable
swap:
     26, Generating copy(T[:ny*nx])
         Generating copyin(Tnew[:ny*nx])
         Generating Tesla code
         27, #pragma acc loop gang /* blockIdx.x */
         28, #pragma acc loop vector(128) /* threadIdx.x */
     28, Loop is parallelizable
```

Result with NVIDIA Volta GPU



500 Jacobi iterations mesh 4096 x 4096

Let’s see the profiler



Unstructured data region

Using structured data regions is not viable in all programs. E.g.:

- Data creation in different contexts
- Object-Oriented programing

OpenACC provides **unstructured** data lifetimes with explicit directives of device memory allocation/deallocation and datar transfer from/to the device.

The programmer must keep track of the data!

Unstructured data region syntax

`#acc enter data [create-clause]`:

- `create:` Allocates memory on the device (no data transfer)
- `copyin:` Allocates memory on the device and copies data from the host to the device

`#acc exit data [exit-clause]`:

- `delete:` Deallocates memory on the device (no data transfer)
- `copyout:` Allocates memory and copies data back to the host

Note: The action performed depends on if the data is already present or not in the device and on its “reference count”.

Use in the Jacobi test case (heat.c)

```
  ...
double * T=NULL, *Tnew=NULL;

allocate(&T,&Tnew,nx,ny);
initConditions(T,nx,ny);

printf("Jacobi algorithm, %d x %d mesh\n", nx, ny);

double st = omp_get_wtime();
while ( error > tol && iter < iter_max )
{
   error = iterJacobi(T, Tnew, nx, ny);
   swap(T, Tnew, nx, ny);

   if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

   iter++;

}
double runtime = omp_get_wtime() - st;

printf(" total: %f s\n", runtime);

deallocate(&T,&Tnew,nx,ny);

   ...
```

Use in the Jacobi test case (jacobi.c)

```
void allocate(double **T, double **Tnew, int nx, int ny){

   double * tt  = (double*)malloc(sizeof(double)*nx*ny);
   double * ttn = (double*)malloc(sizeof(double)*nx*ny);
   *T = tt;
   *Tnew = ttn;
   #pragma acc enter data create(tt[0:nx*ny],ttn[0:nx*ny])

}

void deallocate(double **T, double **Tnew, int nx, int ny){

   double *tt  = *T;
   double *ttn = *Tnew;
   #pragma acc exit data delete(tt[0:nx*ny],ttn[0:nx*ny])
   free(*T);
   free(*Tnew);

}
```

Closing remarks

Closing remarks

In this lecture we have discussed…

GPGPU computing fundamentals

Basic elements of the **CUDA architecture**

What is **OpenACC**

- OpenACC basic syntax and compilation with PGI
- Basic OpenACC directives to expose parallelism
- Basic OpenACC directives for explicit data transfer