Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

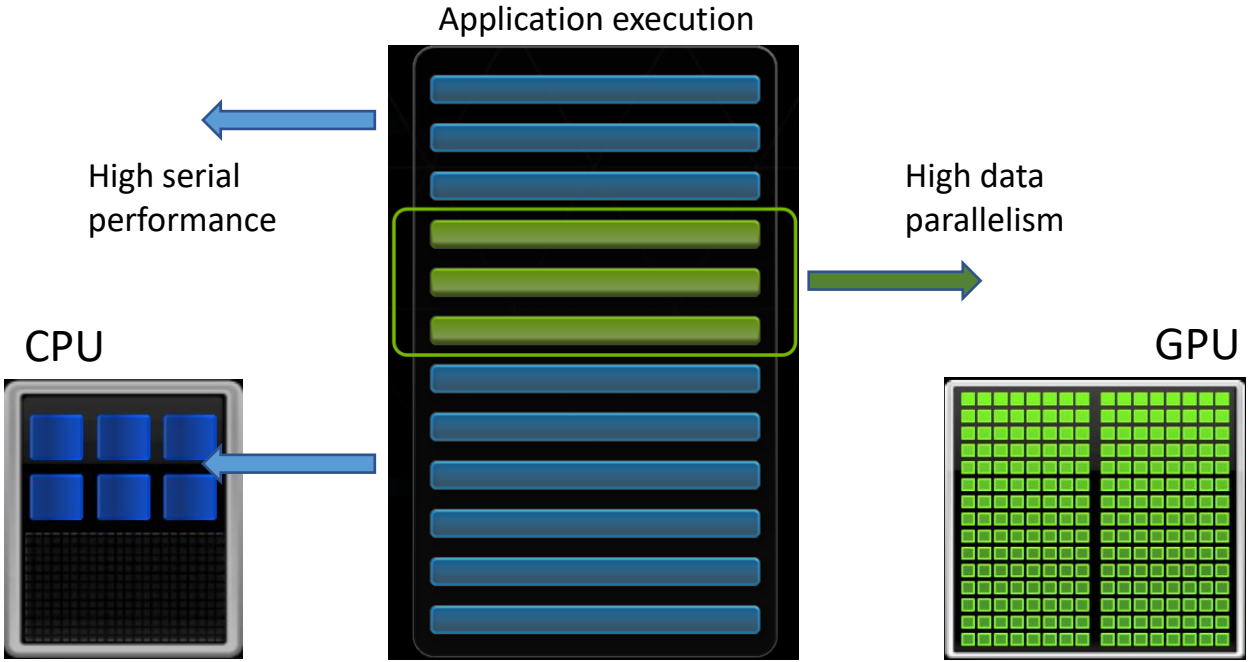
# Accelerated Computing - GPUs and OpenMP

Lecture 7

István Reguly

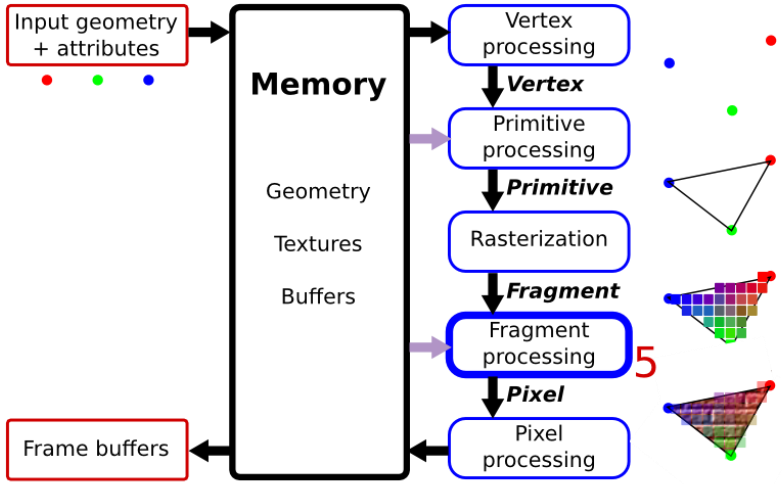
reguly.istvan@itk.ppke.hu

# Accelerated computing



# GPUs

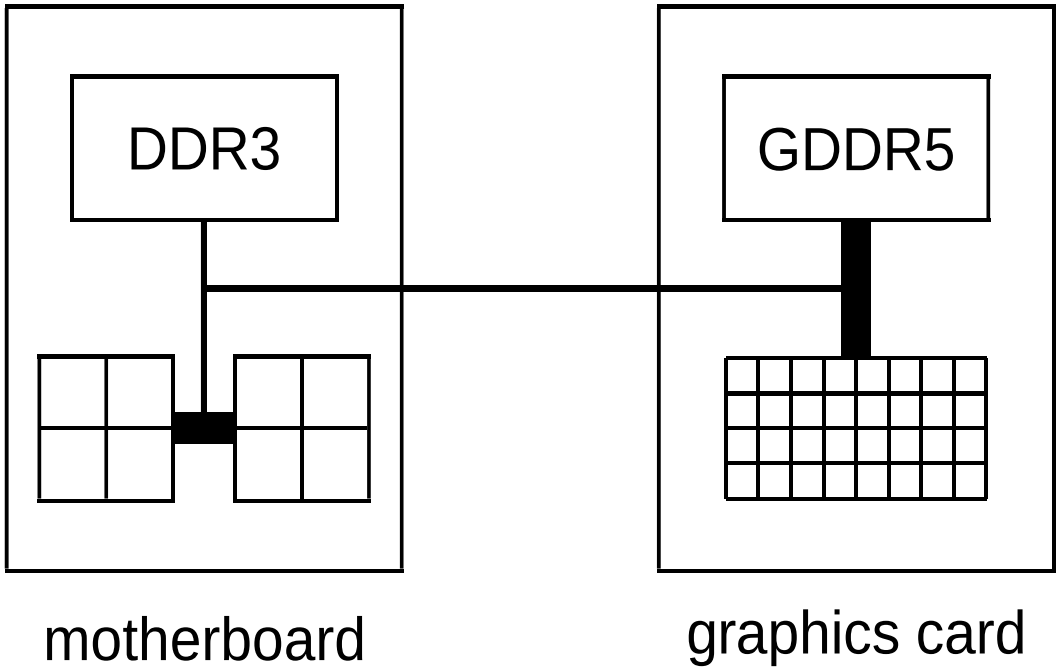
- GPUs are data-parallel computing units
- Origins:
  - Late 90's 3Dfx Voodoo
  - 2000's ATI vs. NVIDIA
  - Graphics oriented, fixed pipelines




# Modern GPUs

- Pipelines became more programmable
  - 2007 – CUDA released
  - General purpose programmability
- Still very much driven by the needs of graphics applications (and AI)
- Increasingly similar to multi-core CPUs with wide vector units

# CPU-GPU system




# GPUs

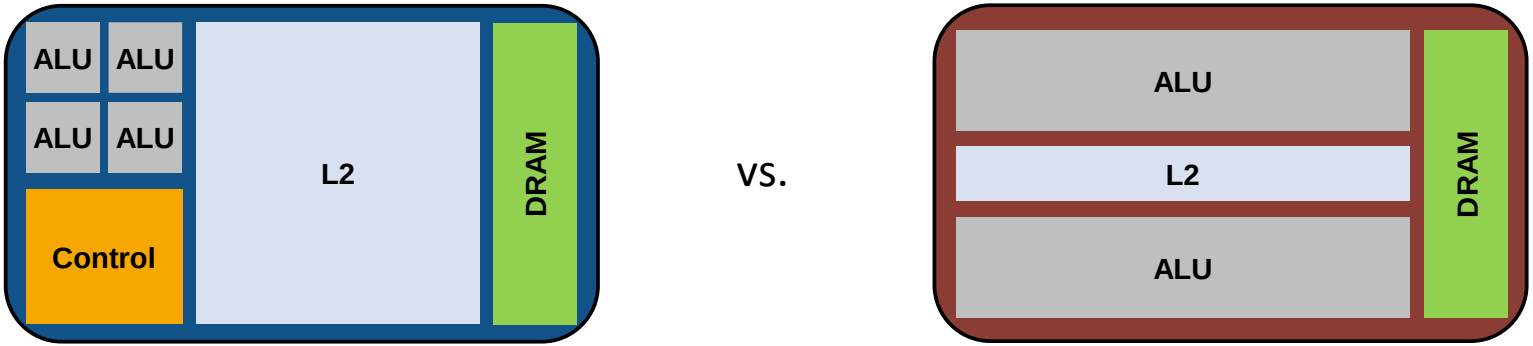
- Up to 10000 “cores” on a chip
- Simplified logic (no out-of-order execution) – most of the chip is devoted to floating-point computation
- Arranged as multiple units with each one effectively being a vector unit, with all cores doing the same thing
- Very high bandwidth to graphics memory (up to 3000 GB/s)
- Not general purpose – graphics and other highly data-parallel applications


# GPU clusters

- People build large supercomputers using GPUs
- One single/dual-socket CPU machine with 1-4 GPUs
  - Like the one you'll be working on
- Interconnected with Infiband – light fibre, with up to 25 Gbit/s (vs. 1 Gbit/s Ethernet)
- Summit – 27.648 GPUs
  - 200 Pflops
  - 13 MW
- Frontier – 37.888 GPUs
  - 1.1 Exaflops
  - 21 MW

# Performance of GPUs

- CPUs are latency oriented: do one thing really fast
  - Caches, Out-of-order execution, etc.
- GPUs are throughput oriented: do a lot of things fast
  - Execution of individual tasks is still slow
  - Relies on massive multi-threading:
    - Up to 16 threads per core
    - While one waits (e.g. for memory), others can work
    - Zero overhead context switching



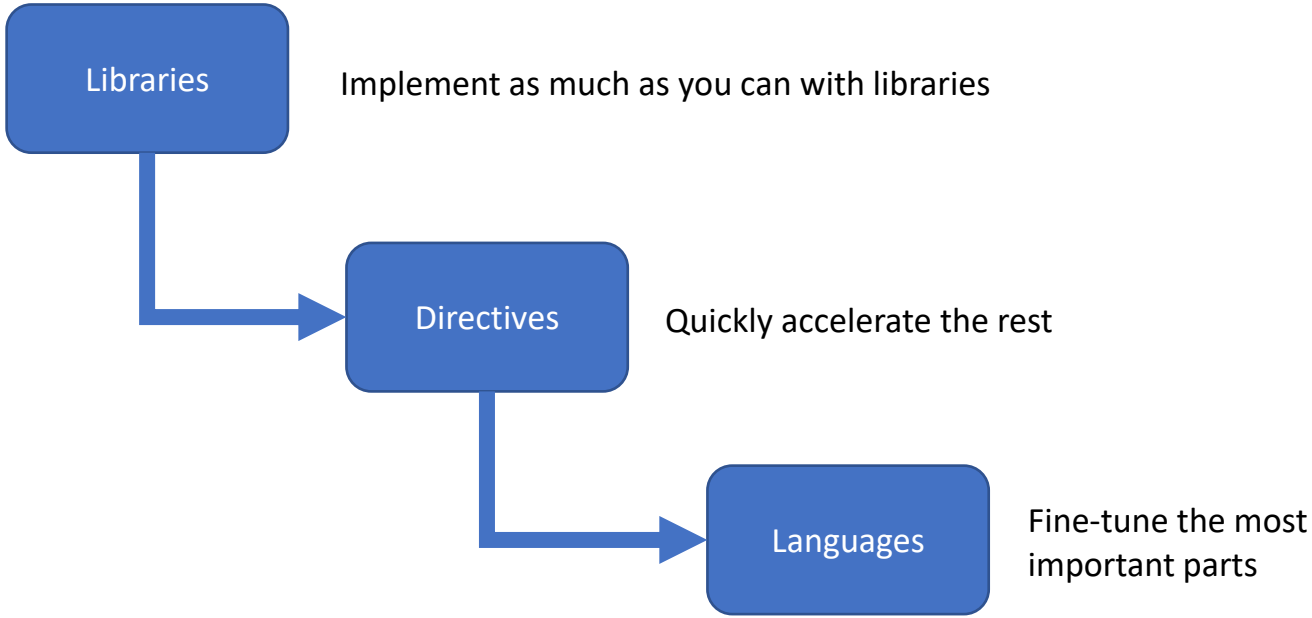

# How do we program GPUs

- GPUs have  a very complicated hardware architecture
  - Significant changes between generations
- To get every last bit of performance, really low level programming is required
  - Before 2007, people used shader languages
  - CUDA was introduced in 2007 – NVIDIA only
  - OpenCL was introduced in 2009 – other vendors too
- These are very low-level approaches!

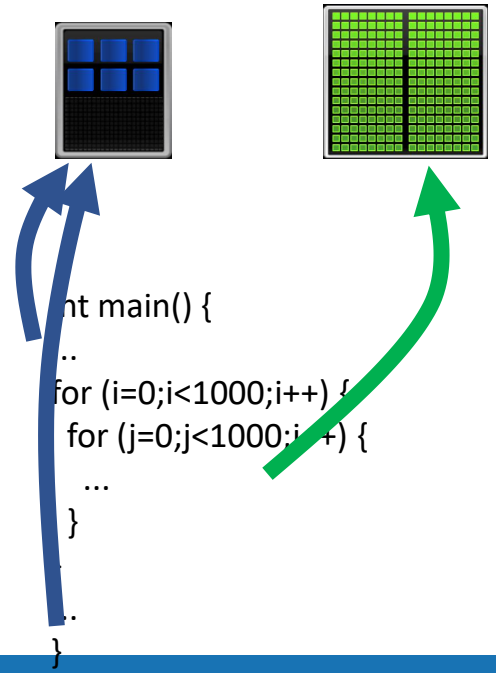
# Simplicity & Performance


- Accelerated Libraries
  - Little or no code change for standard libraries
  - Limited in what the library offers
- Compiler Directives
  - High level: Based on existing languages; simple and familiar
  - High Level: Performance may not be optimal
- Parallel Language Extensions
  - Expose low-level details for maximum performance
  - Often more difficult to learn and more time consuming to implement

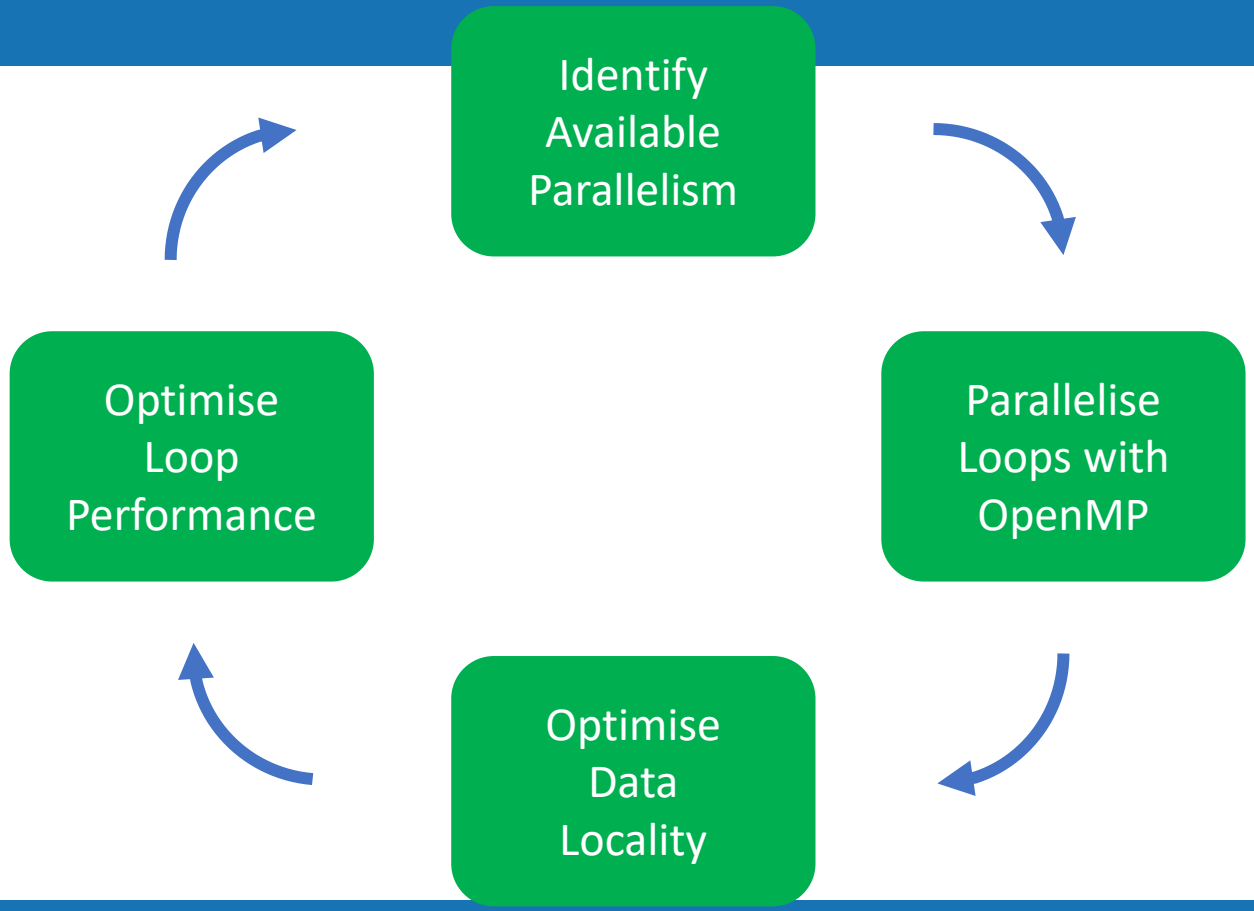
# Code for simplicity and performance



# GPU Directives




- Insert portable compiler directives
- Compiler parallelizes code and manages data movement
- Programmer optimizes incrementally
- Designed for multi-core CPUs, GPUs & many-core Accelerators


Identify Available Parallelism
Parallelise Loops with OpenMP
Optimise Data Locality
Optimise Loop Performance

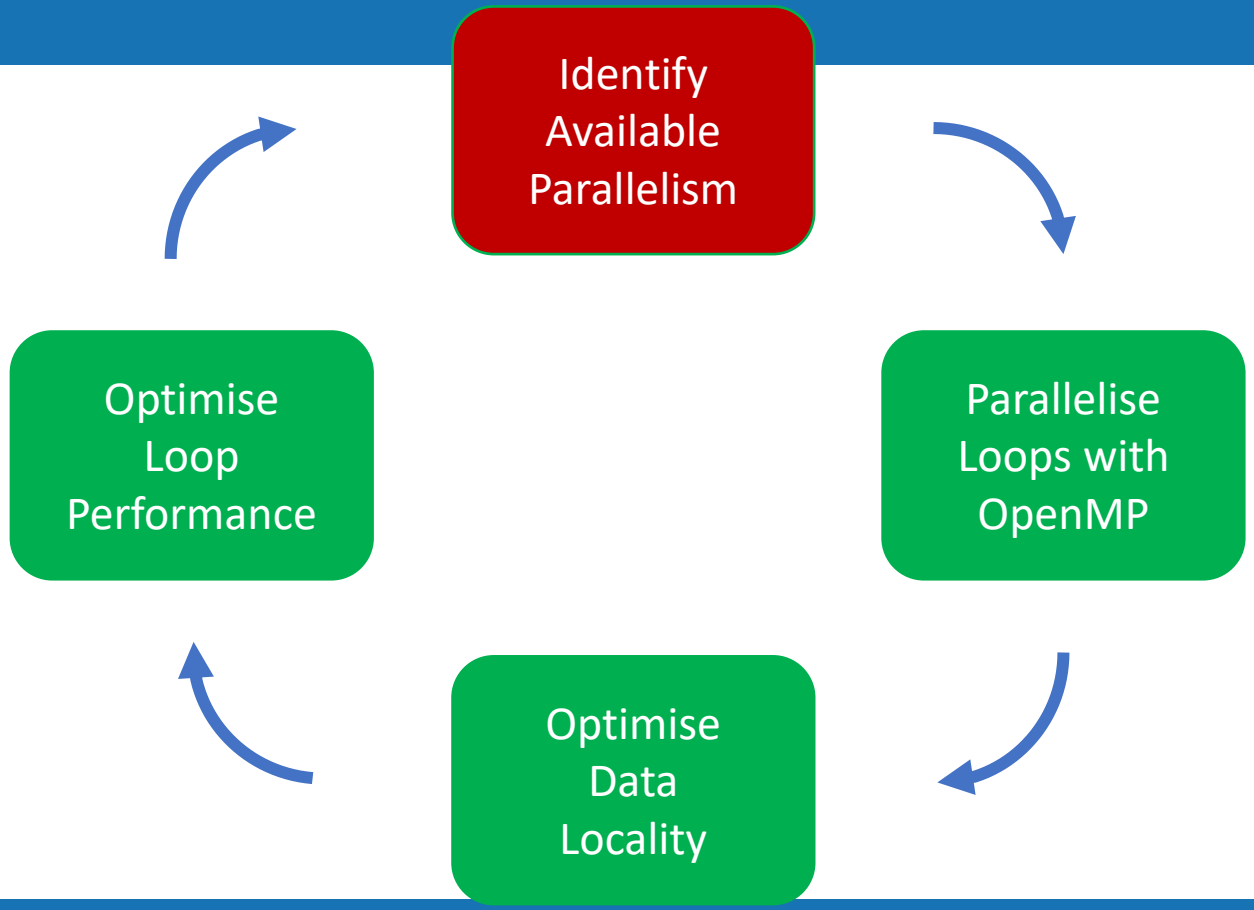
# Our laplace equation solver

```
while ( error > tol && iter < iter_max ) {
 error = 0.0;
 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[j*(imax+2)+i] = 0.25f * (
                                A[(j+1) *(imax+2)+i] +
                                A[(j-1) *(imax+2)+i] +
                                A[j*(imax+2)+i-1] +
                                A[j*(imax+2)+i+1]);
   error = fmax( error, fabs(Anew[...]-A[...]));
  }
 }
 for( int i = 1; i < imax+1; i++ )
  for( int j = 1; j < jmax+1; j++)
   A[j*(imax+2)+i] = Anew[j*(imax+2)+i];
 if(iter % 100 == 0)
  printf("%5d, %0.6f\n", iter, error);
 iter++;
}
```

- Iterate for iter_max
- Iterate across grid
- Compute new value
- Reduce error
- Copy input/output
- Print residual


# Exercise

- Compile the laplace2d.cpp using the Cray compilers
- To compile:
  - `CC -Ofast -fopenmp laplace2d.cpp -o laplace_cpu`
- To run:
  - `salloc -p cpu -c 16 --mem-per-cpu=2000 --time=00:05:00 srun --ntasks=1 ./laplace_cpu`


Identify Available Parallelism
Parallelise Loops with OpenMP
Optimise Data Locality
Optimise Loop Performance

# Identify parallelism

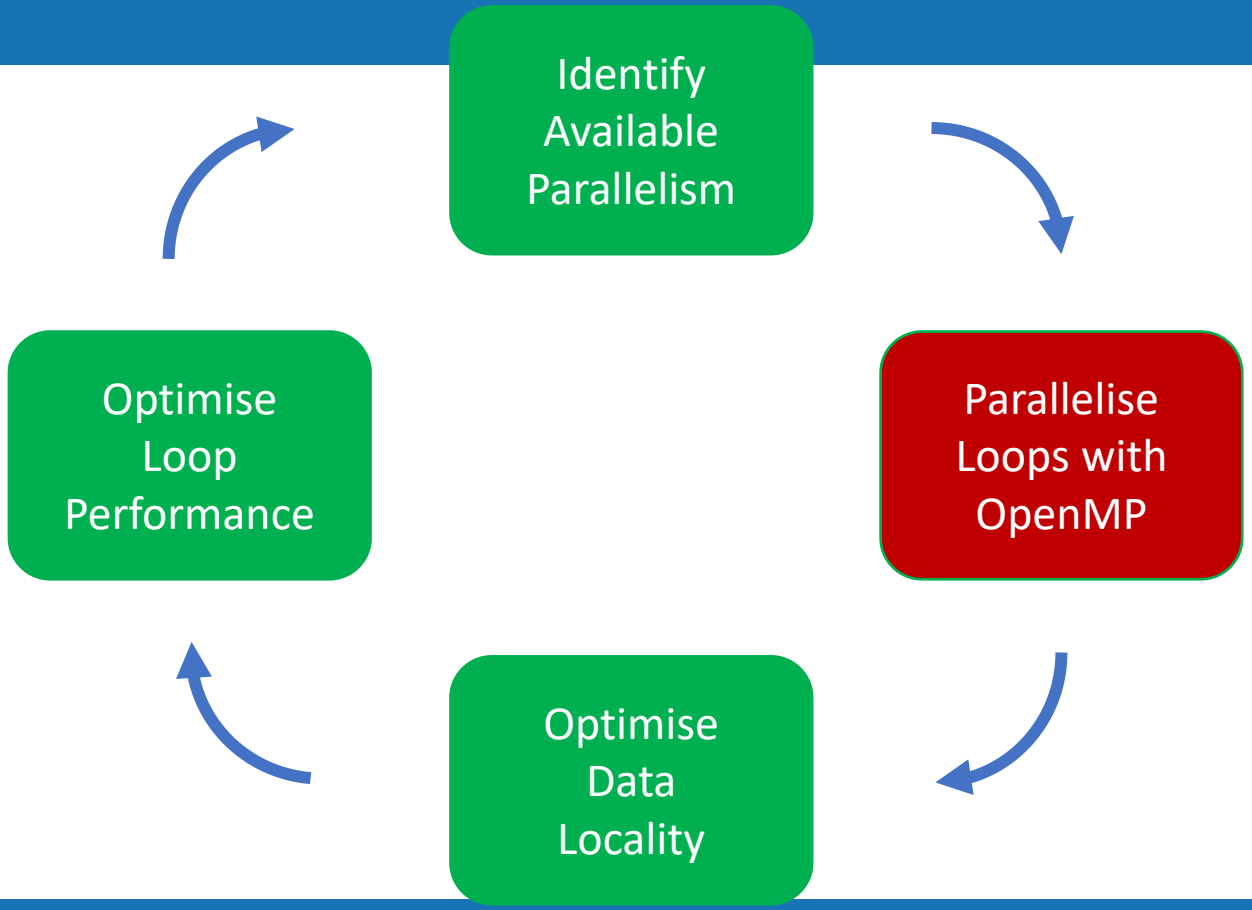
```
while ( error > tol && iter < iter_max ) {
 error = 0.0;
 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[j*(imax+2)+i] = 0.25f * (
                                A[(j+1) *(imax+2)+i] +
                                A[(j-1) *(imax+2)+i] +
                                A[j*(imax+2)+i-1] +
                                A[j*(imax+2)+i+1]);
   error = fmax( error, fabs(Anew[...]-A[...]));
  }
 }
 for( int i = 1; i < imax+1; i++ )
  for( int j = 1; j < jmax+1; j++)
   A[j*(imax+2)+i] = Anew[j*(imax+2)+i];
 if(iter % 100 == 0)
  printf("%5d, %0.6f\n", iter, error);
 iter++;
}
```

Data dependency

Independent iterations

Reduction

Independent iterations


Identify Available Parallelism
Parallelise Loops with OpenMP
Optimise Data Locality
Optimise Loop Performance


# OpenMP directive syntax

- C/C++ - very similar to OpenMP CPU parallelisation

  #pragma omp target [clause [,] clause] ...]

- target – identifies a block of code that should be offloaded to a “target” device - becomes a “kernel”
- teams distribute – chunk up execution between larger “cores” on the GPU
- parallel for – parallelize iterations of for loop across different threads within the “core” on the GPU

```
#pragma target teams distribute parallel for
for(int i=0; i<N; i++) {
    y[i] = a*x[i]+y[i];
}
```

**Kernel**: A function that runs in parallel on the GPU

# Parallelising with OpenMP

```
while ( error > tol && iter < iter_max ) {
 error = 0.0;
#pragma omp target teams distribute parallel for reduction(max:error) \
      map(A[0:(imax+2)*(jmax+2)]) map(Anew[0:(imax+2)*(jmax+2)])
 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[j*(imax+2)+i] = 0.25f * (
                                  A[(j+1) *(imax+2)+i] +
                                  A[(j-1) *(imax+2)+i] +
          A[j*(imax+2)+i-1] +
                                  A[j*(imax+2)+i+1]);
    error = fmax( error, fabs(Anew[...]-A[...]));
  }
 }
#pragma omp target teams distribute parallel for \
      map(A[0:(imax+2)*(jmax+2)]) map(Anew[0:(imax+2)*(jmax+2)])
 for( int i = 1; i < imax+1; i++ )
  for( int j = 1; j < jmax+1; j++)
   A[j*(imax+2)+i] = Anew[j*(imax+2)+i];
 if(iter % 100 == 0)
  printf("%5d, %0.6f\n", iter, error);
 iter++;
}
```

Parallelise loop

Parallelise loop

# Compiling with OpenMP

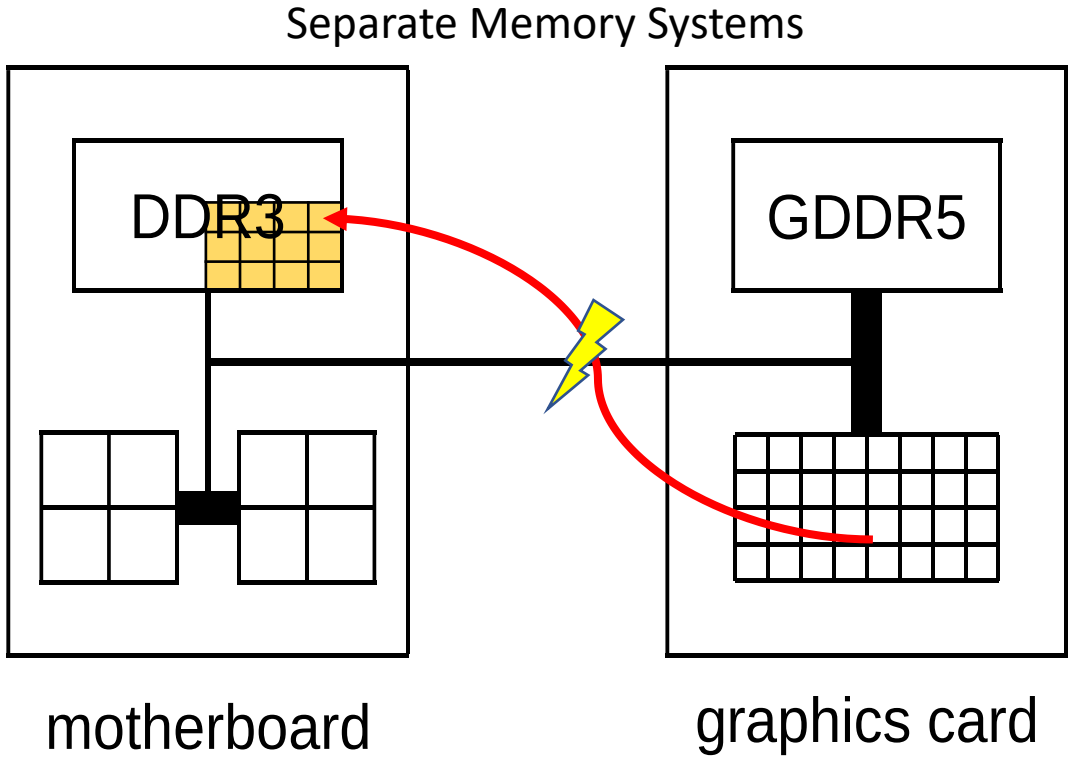
module load craype-accel-nvidia80

module load craype-accel-nvidia80
CC -Ofast laplace2d.cpp -o laplace -fopenmp –lcudart
srun -p gpu --gres=gpu:1 --ntasks=1 --time=00:05:00 --mem=40G ./laplace

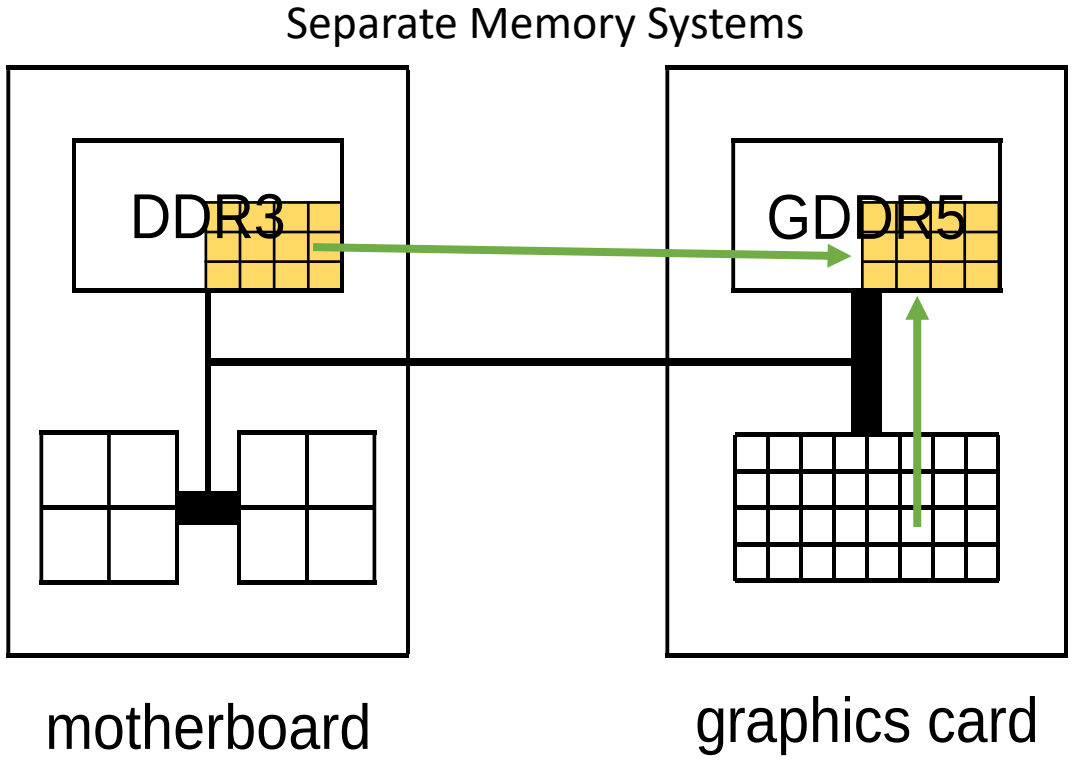
export CRAY_ACC_DEBUG=1
srun -p gpu --gres=gpu:1 --ntasks=1 --time=00:05:00 --mem=40G ./laplace

ACC: Transfer 3 items (to acc 268697672 bytes, to host 0 bytes) from laplace2d.cpp:62
ACC: Execute kernel __omp_offloading_20ea7bcc_c001c75b_main_l62 from laplace2d.cpp:62
ACC: Transfer 3 items (to acc 0 bytes, to host 268697672 bytes) from laplace2d.cpp:62
ACC: Transfer 2 items (to acc 268697664 bytes, to host 0 bytes) from laplace2d.cpp:72
ACC: Execute kernel __omp_offloading_20ea7bcc_c001c75b_main_l72_cce$noloop$form from laplace2d.cpp:72
ACC: Transfer 2 items (to acc 0 bytes, to host 268697664 bytes) from laplace2d.cpp:72

# CPU-GPU system

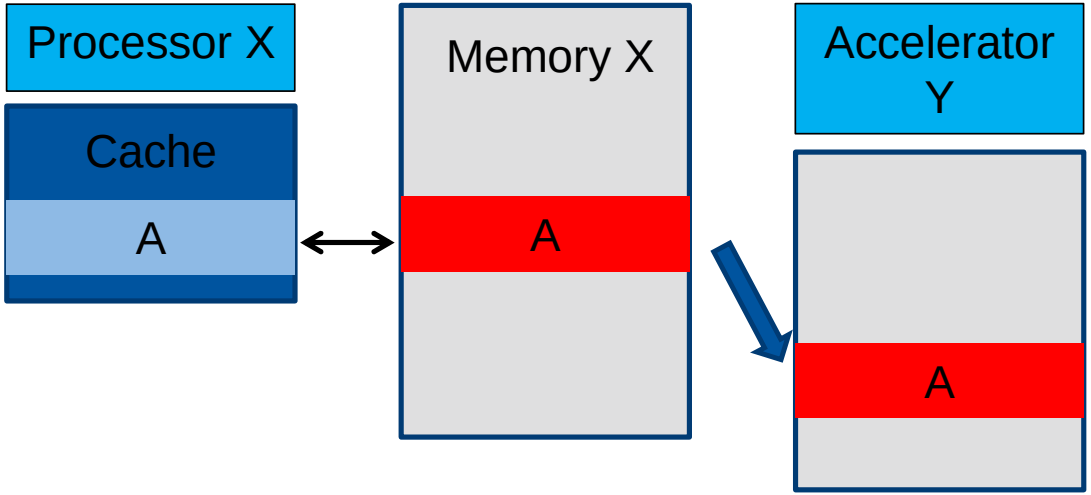


# CPU-GPU system



# CPU and GPU arrays

- OpenMP has to copy our CPU arrays to the GPU
  - But it does not know how big they are
  - Extra clauses on top of acc parallel loop:
  - map(A[0:((imax+2) * (jmax+2))]) map(Anew[0:((imax+2) * (jmax+2))])




# Exercise: Performance

- Measure performance:
  - serial
  - multi-core CPU with OpenMP
  - GPU with OpenMP

- Much slower with OpenMP on GPU

# Why so bad? – Data transfers!

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

A,Anew on host → A,Anew on device

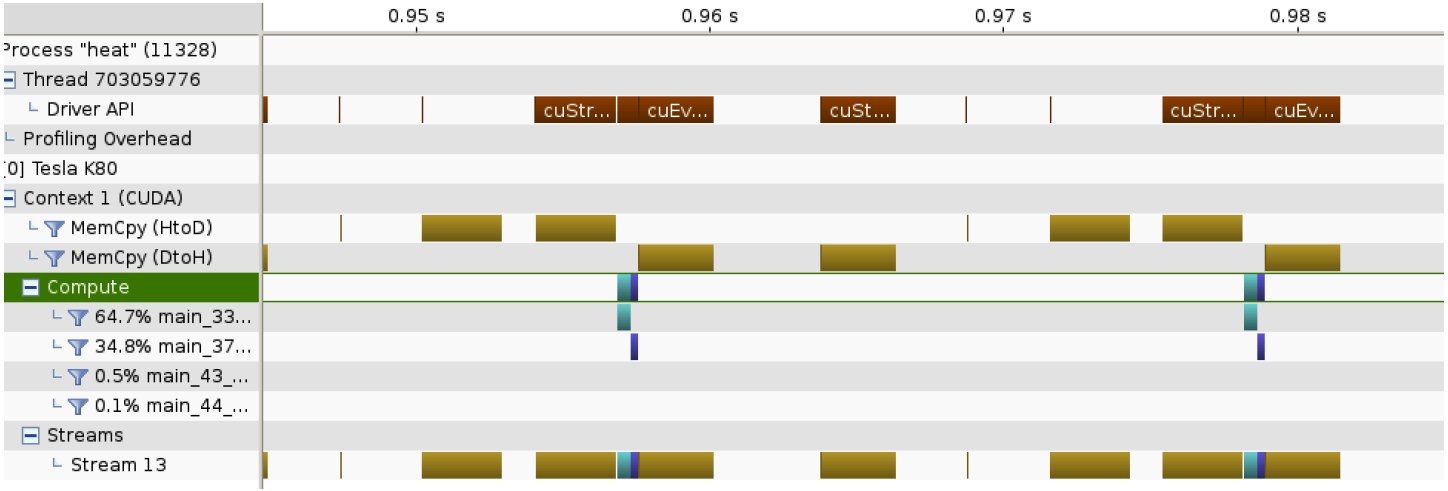
```
#pragma omp target distribute parallel for reduction(max:error) \
      map(A[0:(imax+2)*(jmax+2)]) map(Anew[0:(imax+2)*(jmax+2)])
 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[j*(imax+2)+i] = 0.25f * (
                                A[(j+1) *(imax+2)+i] +
                                A[(j-1) *(imax+2)+i] +
          A[j*(imax+2)+i-1] +
                                A[j*(imax+2)+i+1]);
   error = fmax( error, fabs(Anew[...]-A[...]));
  }
 }
...
```

A,Anew on device → A,Anew on host

**Happens every time!!**

# Profiling the application

- Reduce overall iteration count to ~100
- Use the NVIDIA Visual Profiler (requires X forwarding):
  - nvvp ./laplace2d &



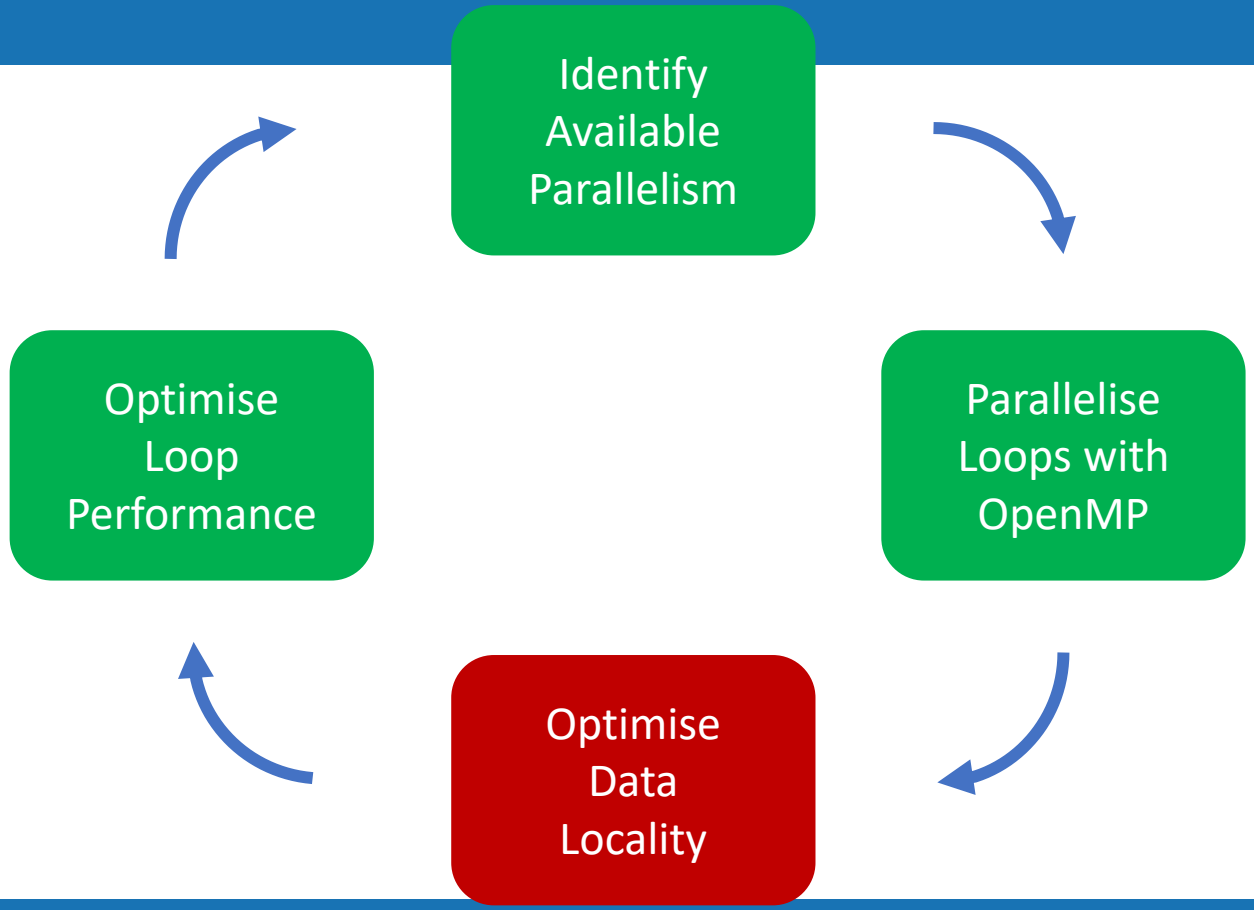
# Identifying data locality

```
while ( error > tol && iter < iter_max ) {
 error = 0.0;
#pragma omp target distribute parallel for reduction(max:error) \
 map(A[0:((imax+2) * (jmax+2))]) \
 map(Anew[0:((imax+2) * (jmax+2))])

 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[j*(imax+2)+i] = 0.25f * (
                                 A[(j+1) *(imax+2)+i] +
                                 A[(j-1) *(imax+2)+i] +
        A[j*(imax+2)+i-1] +
                                 A[j*(imax+2)+i+1]);
   error = fmax( error, fabs(Anew[...]-A[...]));
  }
 }
#pragma omp target distribute parallel for \
map(A[0:((imax+2) * (jmax+2))]) map(Anew[0:((imax+2) * (jmax+2))])
 for( int i = 1; i < imax+1; i++ )
  for( int j = 1; j < jmax+1; j++)
   A[j*(imax+2)+i] = Anew[j*(imax+2)+i];
 if(iter % 100 == 0)
  printf("%5d, %0.6f\n", iter, error);
 iter++;
}
```

Does the CPU need the data between these loop nests?

Does the CPU need the data between iterations?


Identify Available Parallelism
Parallelise Loops with OpenMP
Optimise Data Locality
Optimise Loop Performance


# Defining data regions

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

```
#pragma omp target data
{
#pragma omp target distribute
...
#pragma omp target distribute
...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.


# Data clauses

- `map(list)` - Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- `map(tofrom:list)` – same as above
- `map(to:list)` - Allocates memory on GPU and copies data from host to GPU when entering region.
- `map(from:list)` - Allocates memory on GPU and copies data to the host when exiting region.
- `map(alloc:list)` - Allocates memory on GPU but does not copy


# Array sharing

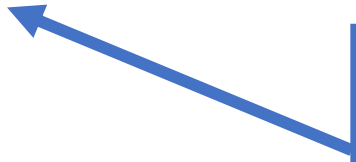
- Some compilers try to determine the size of arrays automatically – check compiler feedback! (NVIDIA nvhpc)
- Sometimes cannot – you have to explicitly specify the shape!

```
#pragma omp target data map(to:a[0:size]) map(from:b[s/4:3*s/4])
```

Note: data clauses can be used on `data`, `target`

# Managing data

```
#pragma omp target data \
 map(A[0:((imax+2) * (jmax+2))]) \
 map(Anew[0:((imax+2) * (jmax+2))])
while ( error > tol && iter < iter_max ) {
 error = 0.0;
#pragma omp target distribute parallel for reduction(max:error)
 for( int i = 1; i < imax+1; i++ ) {
  for( int j = 1; j < jmax+1; j++) {
   Anew[i][j] = 0.25f * ( A[i][j+1] + A[i][j-1]
              + A[i-1][j] + A[i+1][j]);
   error = fmax( error, fabs(Anew[i][j]-A[i][j]));
  }
 }
#pragma omp target distribute parallel for
 for( int i = 1; i < imax+1; i++ )
  for( int j = 1; j < jmax+1; j++)
   A[i][j] = Anew[i][j];
 if(iter % 100 == 0)
  printf("%5d, %0.6f\n", iter, error);
 iter++;
}
```

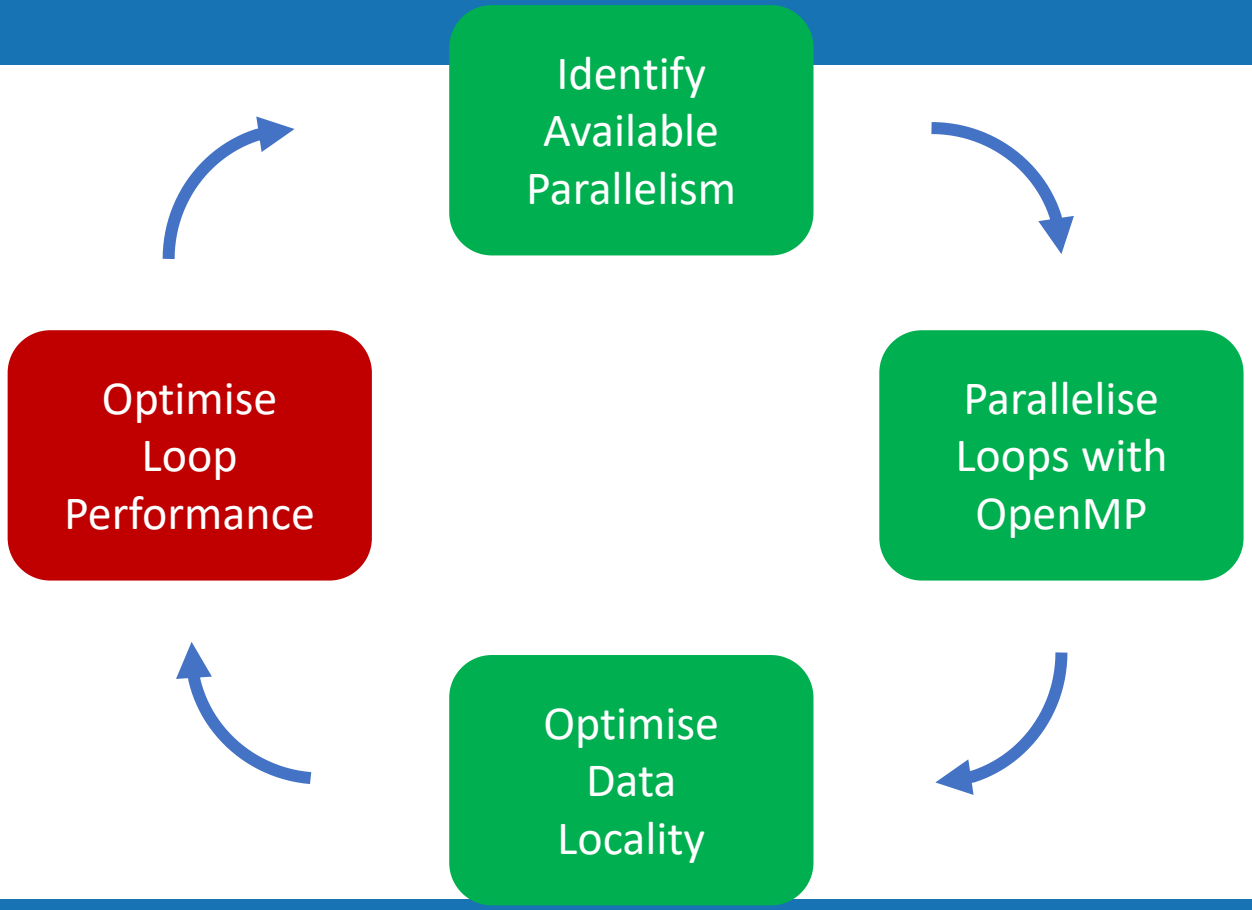


Copy A,Anew to/from the device


# Exercise: Performance

- Add the data region
- What is the performance improvement?
  - Try excluding the data movement operations!
  - Try running for more iterations (max_iters)
  - Try running with a larger mesh!
- What is reported when export CRAY_ACC_DEBUG=1?

- Calculate absolute performance metrics
  - Achieved bandwidth for the application
  - See how achieved bandwidth depends on e.g. problem size


Identify Available Parallelism
Parallelise Loops with OpenMP
Optimise Data Locality
Optimise Loop Performance


# OpenMP `collapse` clause

- `collapse(n)` : Transform the following n tightly nested loops into one, flattened loop.
- Useful when individual loops lack sufficient parallelism or more than 3 loops are nested

```
#pragma omp target distribute parallel for collapse(2)
for(int i=0; i<N; i++)
for(int j=0; j<N; j++) ...
```

```
#pragma omp target distribute parallel for
for(int ij=0; ij<N*N; ij++)
...
```


# Exercise

- Open Lab 7 cg.cpp
- Parallelise with OpenMP CPU/GPU
  - Use collapse wisely!
  - Plenty of reductions
- Compare performance on CPU vs. GPU


# Homework – due Apr 21

- Add OpenMP GPU offload to our lbm_d2q9 project
  - Analogous to CPU side OpenMP
  - Use data regions