UNIVERSITÉ DU LUXEMBOURG

High Performance Computing & Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

LUXEMBOURG
LET'S MAKE IT HAPPEN

Uni.lu HPC School 2021

PS10b: Introduction to GPU programming with OpenAcc and OpenCL

Uni.lu High Performance Computing (HPC) Team

L. Koutsantonis, T. Carneiro

University of Luxembourg (UL), Luxembourg

https://hpc.uni.lu/

Objectives

The objective of this tutorial is to show how the OpenAcc directives can be used to accelerate a numerical solver commonly used in engineering and scientific applications. After completing the exercise of this tutorial, you would be able to:

- Transfer data from host to device using the data directives,
- Accelerate a nested loop application with the loop directives, and,
- Use the reduction clause to perform summation on variables or elements of a vector.

The Jacobi method

- Iterative method for solving a system of equations:

$$Ax = b$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

where, the elements a_{ij} and b_j are constants and x_j are the unknowns.

- At each iteration, the elements x_i are updated using their previous estimations by:

$$x_i^k = \frac{1}{a_{ii}} \left(b_i - \sum_{i \neq j} a_{ij} x_j^{k-1} \right)$$

The Jacobi method: Termination

- The convergence of the algorithm towards the solution is monitored through an error function calculated at each step on the current and previous estimates. Unusually, this function is the sum of squared differences:

$$Error = \sum_i (x_i^k - x_i^{k-1})^2$$

- The algorithm terminates when this error reaches a desired threshold:

$$Error < Thres$$

The Laplace Equation

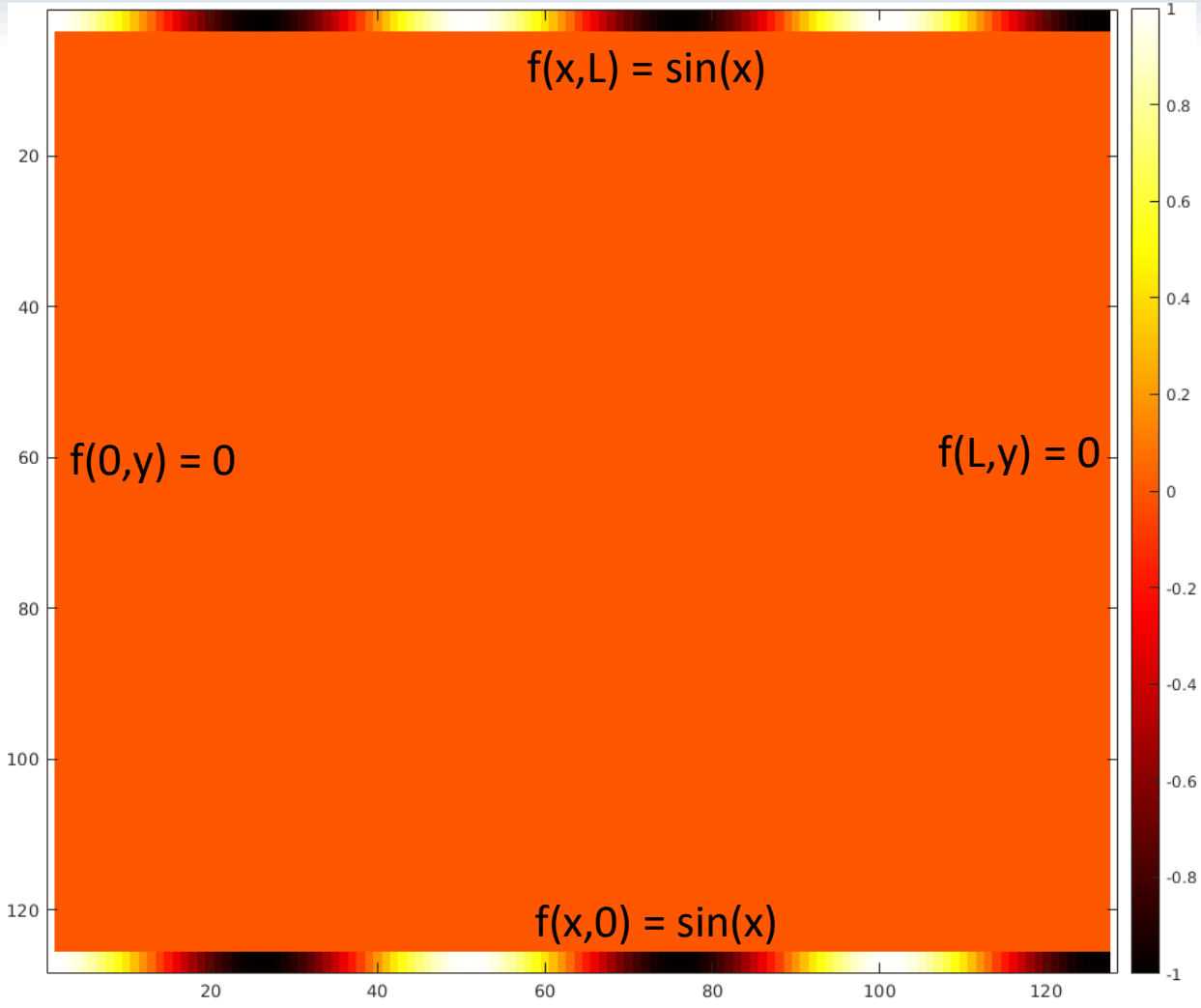
- The Laplace differential equation in 2D is given by:

$$\nabla f = 0 \coloneqq \frac{d^2 f}{dx^2} + \frac{d^2 f}{dy^2} = 0$$

- It models the steady state of a distribution (e.g. Temperature) is 2D space.
- The Laplace differential equation can be solved using the Jacobi method if we know the boundary conditions (e.g. the temperature at the edges of the physical region of interest)

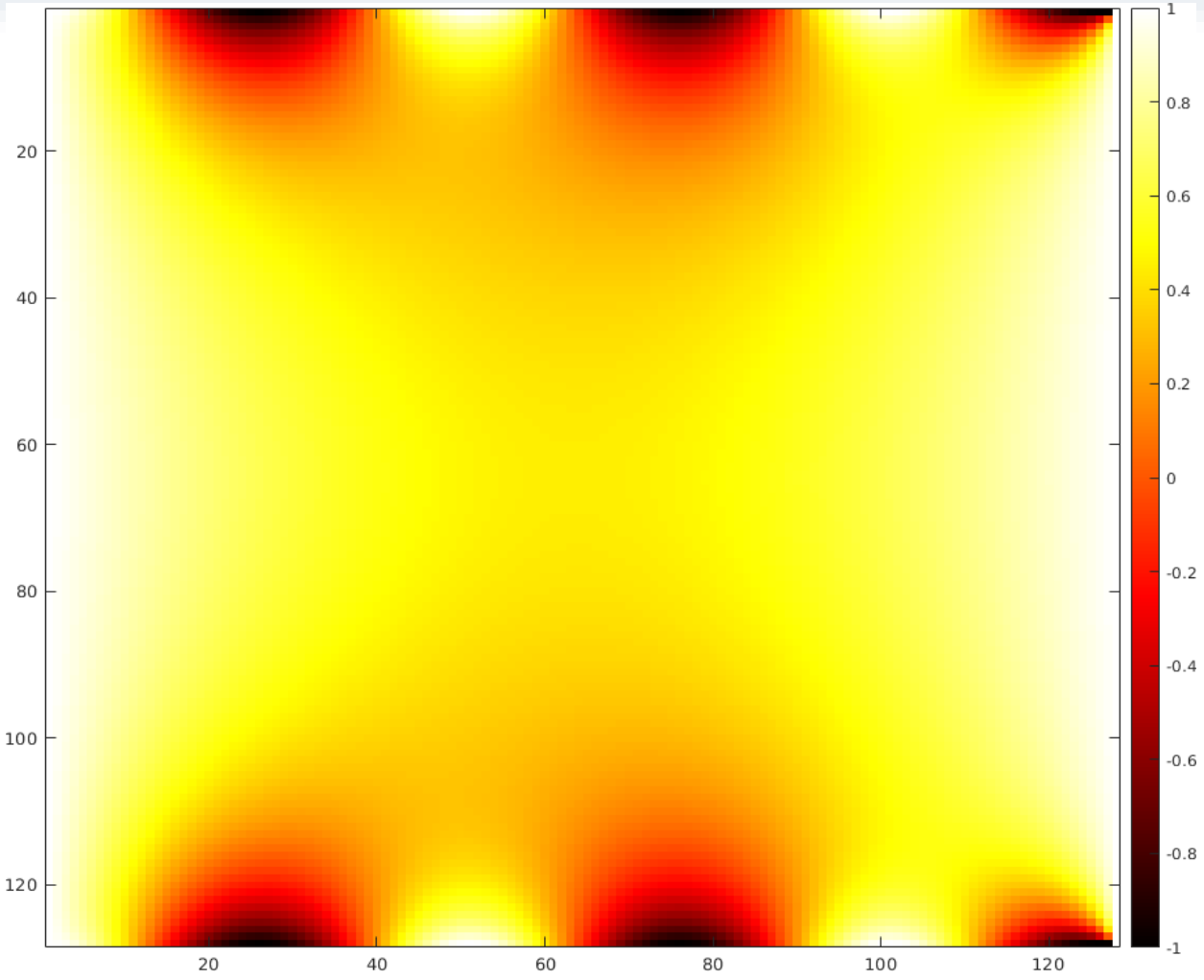
The Laplace Equation

Boundary Conditions



The Laplace Equation

Solution Satisfying the boundary conditions obtained with Jacobi



The Laplace Equation

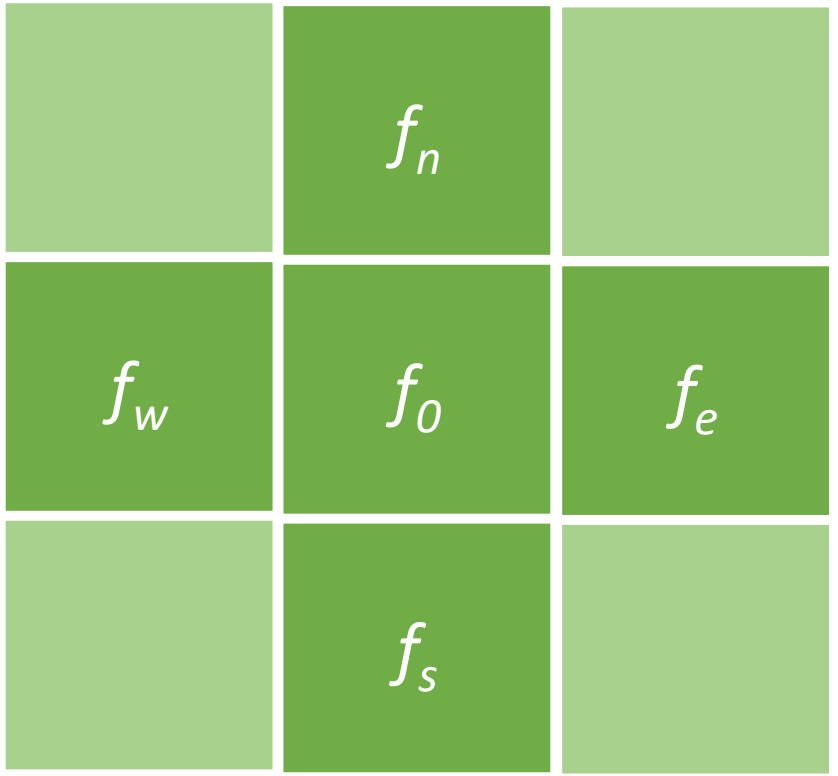
Derivation of *the linear system of equations* with numerical derivatives

$$\frac{d^2f}{dx^2} = \frac{1}{\delta^2}(f_e - 2f_0 + f_w)$$

$$\frac{d^2f}{dy^2} = \frac{1}{\delta^2}(f_n - 2f_0 + f_s)$$

$$\boldsymbol{f_0 = \frac{1}{4}(f_n + f_s + f_e + f_w)}$$

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1` & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix}$$



Implementation of Jacobi Method in C

- calcTempStep (T, Tnew, n, m):
  - Calculates the new values from neighbors (stored in T) and stores the result in a new matrix (Tnew). Size of T is n×m.
  - Computes and returns the error for monitoring the convergence.
- Update(T, Tnew, n,m)
  - Update T with the values stored in Tnew (Tnew(i,j) = T(i,j))
- Terminate when error becomes smaller than a predefined threshold

```
while ((iter < miter )&& (error > thres)){

      error = calcTempStep(T, Tnew, n, m);
      update(T, Tnew, n, m);

      if(iter % 50 == 0) printf("Iterations = %5d, Error = %16.10f\n", iter, error);

      iter++;
  }
```

Implementation of Jacobi Iteration in C

```
float calcTempStep(float *restrict F, float *restrict Fnew, int n, int m){
        float Fu, Fd, Fl, Fr;
        float error = 0.0;

        for (int i = 1; i < n-1; i++){
                for (int j = 1; j < m-1; j++){
                        Fu = F[(i-1)*m + j];
                        Fd = F[(i+1)*m + j];
                        Fl = F[i*m + j - 1];
                        Fr = F[i*m + j + 1];
                        Fnew[i*m+j] = 0.25*(Fu + Fd + Fl + Fr);
                        error += (Fnew[i*m+j] - F[i*m+j])*(Fnew[i*m+j] - F[i*m+j]);
                        }
                }
        return error;
}
```

For each central value find the neighbors and calculate the new value (nested loop).

Sum ovel the matrix element the calculated error (Reduction)

Implementation of Jacobi Iteration in C

```
void update(float *restrict F, float  *restrict Fnew, int n, int m){

        for (int i = 0; i < n; i++)
                for (int j = 0; j < m; j++ )
                        F[i*m+j] = Fnew[i*m+j];

}
```

Update T with Tnew (nested loop)

Exercise: Parallelize Jacobi iteration with OpenAcc

1. Use the data directive to load T and Tnew in device memory before the loop in the function `main`.

```
//1. Code Here! (1 line)
while ((iter < miter )&& (error > thres)){

        error = calcTempStep(T, Tnew, n, m);
        update(T, Tnew, n, m);

        if(iter % 50 == 0) printf("Iterations = %5d, Error = %16.10f\n", iter,
error);

        iter++;
    }
```

Exercise: Parallelize Jacobi iteration with OpenAcc

2. Use the parallel loop directive with a reduction clause to parallelize the nested loop in function `calcTempStep`. (Reduction in error)

```
float calcTempStep(float *restrict F, float *restrict Fnew, int n, int m){
        float Fu, Fd, Fl, Fr;
        float error = 0.0;
        //2. Code Here! (1 line)
        for (int i = 1; i < n-1; i++){
                //3. Code Here! (1 line)
                for (int j = 1; j < m-1; j++){
                        Fu = F[(i-1)*m + j];
                        Fd = F[(i+1)*m + j];
                        Fl = F[i*m + j - 1];
                        Fr = F[i*m + j + 1];
                        Fnew[i*m+j] = 0.25*(Fu + Fd + Fl + Fr);
                        error += (Fnew[i*m+j] - F[i*m+j])*(Fnew[i*m+j] - F[i*m+j]);
 …
```

Exercise: Parallelize Jacobi iteration with OpenAcc

3. Use the parallel loop directive to parallelize the nested loop in function `update`.

```
void update(float *restrict F, float  *restrict Fnew, int n, int m){
        //3. Code Here! (1 line)
        for (int i = 0; i < n; i++)
                //3. Code Here! (1 line)
                for (int j = 0; j < m; j++ )
                        F[i*m+j] = Fnew[i*m+j];

}
```

Access to the Data

 # Clone Tutorials github repository

- (access)$> **cd** ~/git/github.com/ULHPC/tutorials/
- (access)$> git pull
- (access)$> **cd** OpenAccExe/exercise/

- ***Source File "jacobi.c"***

Compilation and Run

1. Reserve an interactive job/allocate GPU resources

```
srun --reservation=hpcschool-gpu -p gpu --ntasks-per-node 1 -c7 -G 1 --pty bash
```

2. Load the PGI compiler

```
module load compiler/PGI/19.10-GCC-8.3.0-2.32
```

3. Compile the code

```
pgcc -acc –ta=nvidia -Minfo jacobi.c -o jacobi
```

4. Run

```
./jacobi
```

Compiler Output

```
calcTempStep:
    41, Generating copyin(F[:n*m]) [if not already present]
        Generating Tesla code
        42, #pragma acc loop gang /* blockIdx.x */
            Generating reduction(+:error)
        44, #pragma acc loop vector(128) /* threadIdx.x */
            Generating reduction(+:error)
    41, Generating implicit copy(error) [if not already present]
        Generating copyout(Fnew[:n*m]) [if not already present]
    42, FMA (fused multiply-add) instruction(s) generated
    44, Loop is parallelizable
update:
    65, Generating copyin(Fnew[:n*m]) [if not already present]
        Generating copyout(F[:n*m]) [if not already present]
        Generating Tesla code
        67, #pragma acc loop gang /* blockIdx.x */
        69, #pragma acc loop vector(128) /* threadIdx.x */
    69, Loop is parallelizable
        Memory copy idiom, loop replaced by call to __c_mcopy4
main:
   127, Generating copy(Tnew[:m*n],T[:m*n]) [if not already present]
```

Solution: Parallelize Jacobi iteration with OpenAcc

1. Use the data directive to load T and Tnew in device memory before the loop in the function `main`.

```
#pragma acc data copy(T[0:m*n])  copy(Tnew[0:m*n])
while ((iter < miter )&& (error > thres)){

        error = calcTempStep(T, Tnew, n, m);
        update(T, Tnew, n, m);

        if(iter % 50 == 0) printf("Iterations = %5d, Error = %16.10f\n", iter,
error);

        iter++;
    }
```

Solution: Parallelize Jacobi iteration with OpenAcc

2. Use the parallel loop directive with a reduction clause to parallelize the nested loop in function `calcTempStep`. (Reduction in error)

```
float calcTempStep(float *restrict F, float *restrict Fnew, int n, int m){
        float Fu, Fd, Fl, Fr;
        float error = 0.0;
#pragma acc parallel loop  reduction(+:error) copyin(F[0:m*n])  copyout(Fnew[0:m*n])
        for (int i = 1; i < n-1; i++){
#pragma acc loop reduction(+:error)
                for (int j = 1; j < m-1; j++){
                        Fu = F[(i-1)*m + j];
                        Fd = F[(i+1)*m + j];
                        Fl = F[i*m + j - 1];
                        Fr = F[i*m + j + 1];
                        Fnew[i*m+j] = 0.25*(Fu + Fd + Fl + Fr);
                        error += (Fnew[i*m+j] - F[i*m+j])*(Fnew[i*m+j] - F[i*m+j]);
 …
```

Solution: Parallelize Jacobi iteration with OpenAcc

3. Use the parallel loop directive to parallelize the nested loop in function `update`.

```
void update(float *restrict F, float  *restrict Fnew, int n, int m){
#pragma acc parallel loop copyin(Fnew[:(m*n)]) copyout(F[:(m*n)])
        for (int i = 0; i < n; i++)
#pragma acc loop
                for (int j = 0; j < m; j++ )
                        F[i*m+j] = Fnew[i*m+j];

}
```

Results

Serial Version

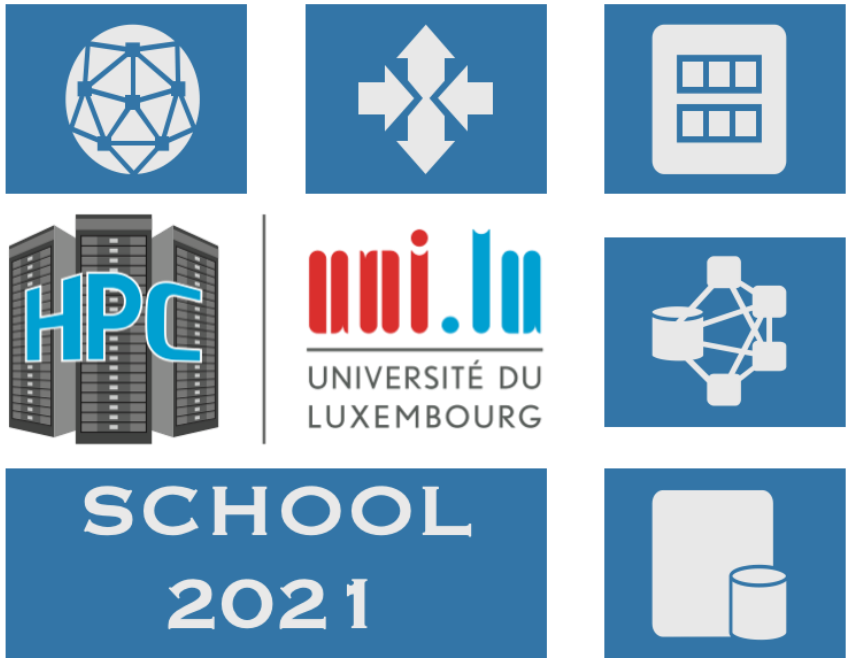
```
Total Iterations =        10000
Error            =    0.0000860624
Total time (sec) =       64.332
```

OpenAcc

```
Total Iterations =        10000
Error            =    0.0000861085
Total time (sec) =       0.808
```

80 times faster

Thank you for your attention!



University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
mail: hpc@uni.lu

https://hpc.uni.lu/

uni.lu
UNIVERSITÉ DU LUXEMBOURG