David Gutzwiller, NUMECA USA
(david.gutzwiller@numeca.com)
Dr. Ravi Srinivasan, Dresser-Rand
Alain Demeulenaere, NUMECA USA
5/9/2017

GTC 2017
S7672

OpenACC Best Practices:
Accelerating the C++ NUMECA FINE/Open CFD Solver

Goal

Adapt the FINE/Open CFD solver for execution in a heterogeneous CPU + GPU environment, immediately targeting the OLCF Titan supercomputer, but aiming for general platforms

18,688 compute nodes, each containing:

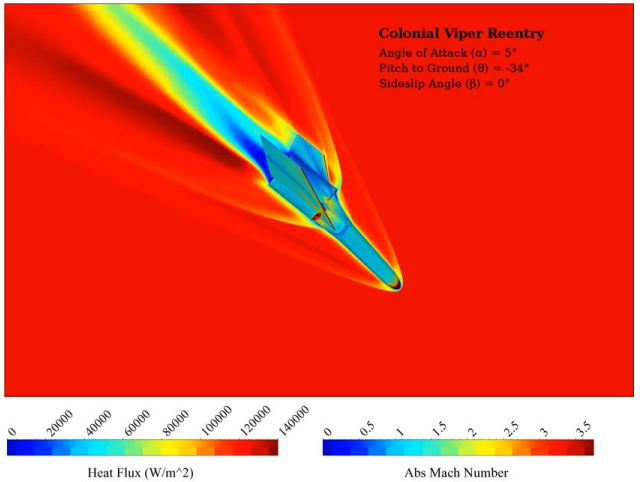
- ➔ 1 AMD Opteron 6284 CPU
- ➔ 1 NVIDIA Tesla K20X GPU
- ➔ 32 GB main memory, 6 GB GPU memory
- ➔ PCIe Gen2

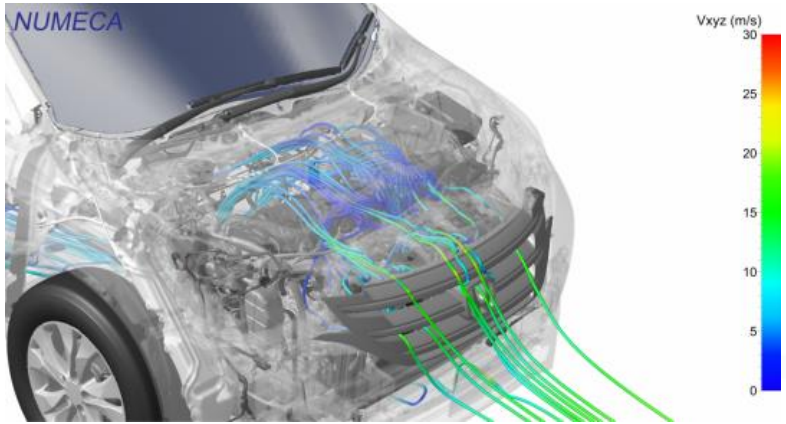
Does it work: Yes!
~1.5X-2X Global speedup relative to CPU only execution
How?

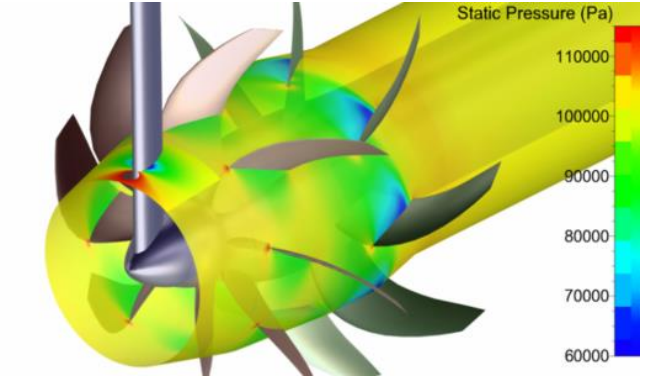
NUMECA FINE/Open

FINE/Open CFD Solver

- ➔ Three dimensional, unstructured, general purpose CFD solver.
- ➔ Supports mixed element and non-conformal mesh topologies.
- ➔ One tool for many classes of fluid flow problems.
  - ◆ Internal and external flows.
  - ◆ Incompressible and compressible fluids.
  - ◆ Low speed to hypersonic.
  - ◆ Advanced chemistry and combustion.
  - ◆ Fluid structure interaction (FSI).
- ➔ High-fidelity simulation for industries including automotive, aerospace, turbomachinery, and power generation.

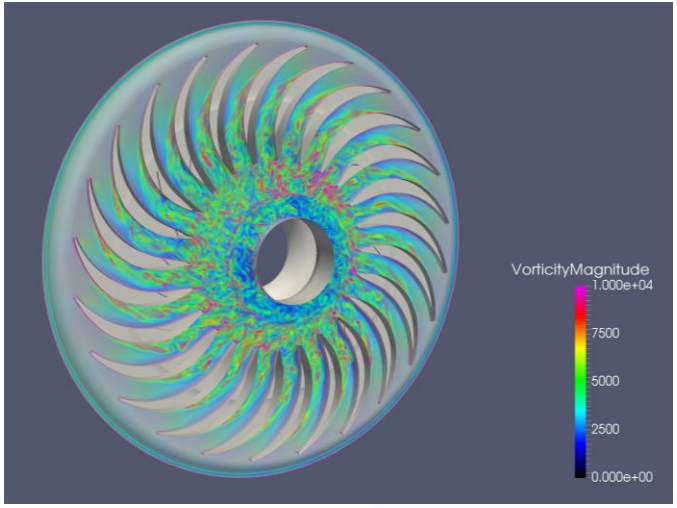


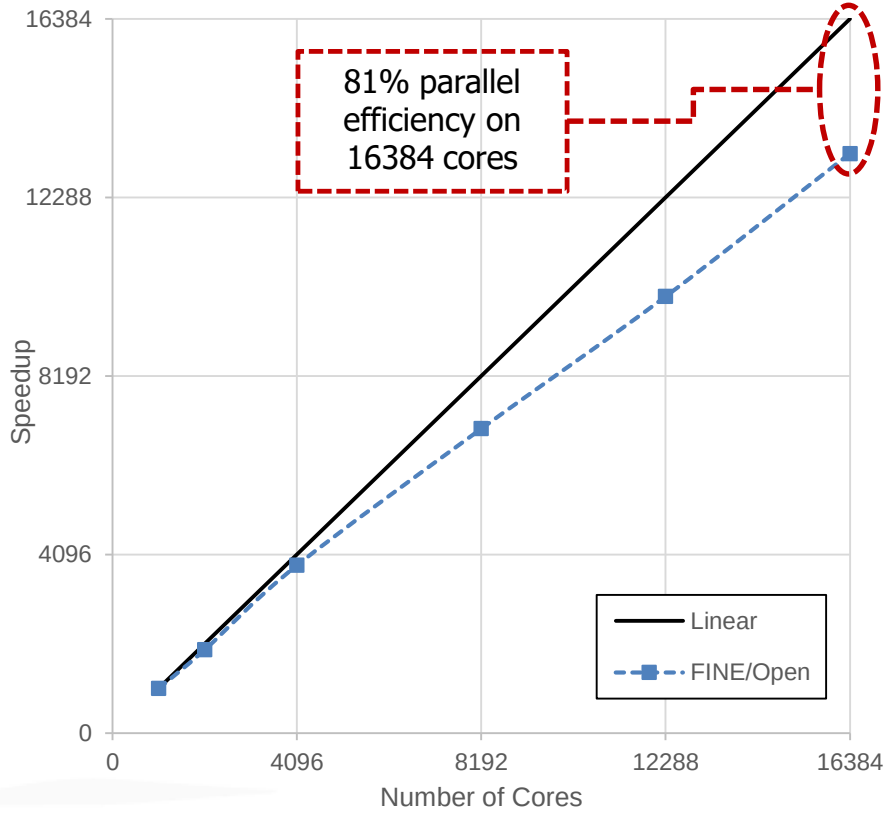




Existing FINE/Open Programming Model

- ➔ Written in C++, following an object oriented programming model.
- ➔ Flat-MPI parallel implementation.
- ➔ Parallel I/O with the CGNS V3 library.
- ➔ Demonstrated scalability on up to 20,000 cores.





GPU Acceleration Strategy

Industrial Constraints

- ➔ For maintainability, duplicate code should be minimized.
- ➔ Large changes to data structures should be avoided.
- ➔ Portability across multiple operating systems and with different hardware is a priority.
- ➔ GPU acceleration should never result in a solver slowdown, regardless of the hardware or dataset.

Incremental Acceleration with OpenACC Directives

- ➔ Identify high cost routines and instrument with OpenACC directives, offloading solver "hot spots" to the GPU.
- ➔ Minimize data transfer.
- ➔ Implement a technique for runtime tuning of the acceleration.

Mandatory OpenACC Functionality

- ➔ Efficient deep copy of non-rectangular arrays with multiple layers of pointer indirection (<T>**).
- ➔ Multiple layers of nested method or function calls within accelerated loops.
- ➔ Thread safe atomic operations.
- ➔ Efficient copy of arrays of (flat) structures.
- ➔ Unstructured data regions.

To be explored in detail

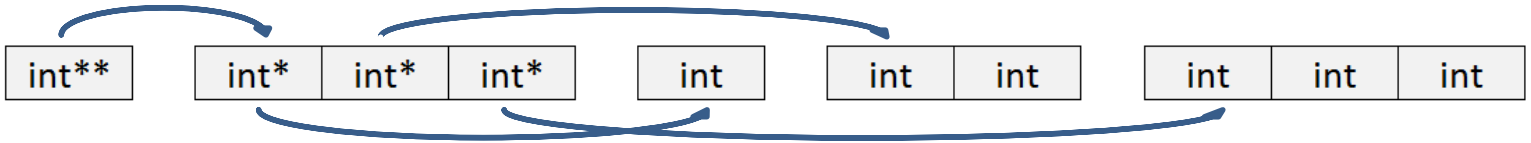
OpenACC
Directives for Accelerators

Deep Copy of Non-Rectangular Arrays

Problem

- ➔ Some data in the FINE/Open solver is stored in arrays allocated on the heap with multiple layers of pointer indirection (T**, pointers to pointers).
- ➔ The layout of these arrays is unchanged over the length of a computation.
- ➔ These arrays may be "jagged", non-rectangular in shape and may even contain NULL pointers.
- ➔ Pointers to these arrays are passed extensively throughout the sources.
- ➔ Offloading the data structures to the GPU as-is would be inefficient, requiring a separate data transfer for each guaranteed contiguous block of memory, and reconstruction of the pointer tree in GPU memory.

```
int nbRow  = 3;
int** data = new int*[nbRow];
for (int i=0; i<nbRow; i++)
{
   int nbElem = i+1;
   data[i] = new int[nbElem];
}
```

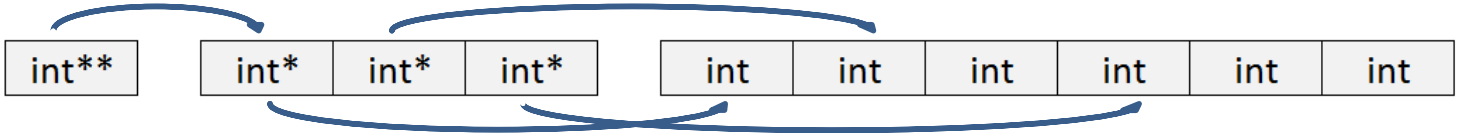


Deep Copy of Non-Rectangular Arrays

Solution

- ➔ Wrap the problematic data structures in a linearized array class. **AccArray<dataType, nbDim>**
- ➔ All data is stored in a single contiguous array, pre-sized based on the total size of the data.
- ➔ The class maintains its own array of pointers allowing unchanged [][] array access throughout the solver.
- ➔ Code modifications only needed at the point of data allocation and in the limited number of accelerated loops.

```
int nbRow  = 3;
AccArray<int> dataAcc* = new AccArray<int>;
for (int i=0; i<nbRow; i++)
{
   int nbElem = i+1;
   dataAcc->pushRow(nbElem);
}
dataAcc->allocateSpace();
dataAcc->createDevice();
int** data = dataAcc->getPointers();
```



Deep Copy of Non-Rectangular Arrays

Device Pointer Tree Setup – 2 Methods

```
void createDeviceSlow()
{
  acc_copyin(this,sizeof(*this));
  acc_copyin(_data,_dataSize*sizeof(T));
  acc_create(_dataPointers,_nbRow*sizeof(T*));
  acc_attach((void**)&_dataPointers);
  for (int iRow=0; iRow<_nbRow; iRow++)
  {
    acc_attach((void**)&_dataPointers[iRow]);
  }
}

void createDeviceFast()
{
  acc_copyin(this,sizeof(*this));
  acc_copyin(_data,_dataSize*sizeof(T));
  acc_copyin(_dataOffsets,_dataSize*sizeof(int));
  acc_create(_dataPointers,_nbRow*sizeof(T*));
  acc_attach((void**)&_dataPointers);
  #pragma parallel loop present(_dataPointers,_dataOffsets,_data)
  for (int iRow=0; iRow<_nbRow; iRow++)
  {
    _dataPointers[i] = &(_data[_dataOffsets[i]]);
  }
}
```

NVIDIA K40, PCIe3
AccArray with 1M rows, 1-5 elements per row.
CreateDeviceSlow(): 9.94s
CreateDeviceFast(): 0.23s

There is significant overhead to looped **acc_attach()** operations. Alternatively, offload metadata to the device and update the pointers on the device directly.

Deep Copy of Non-Rectangular Arrays

Data Update Methods

```
    ...

    void updateDevice()
    {
      acc_update_device(_data,_dataSize*sizeof(T));
    }

    void updateHost()
    {
      acc_update_self(_data,_dataSize*sizeof(T));
    }

    void updateDevice(int iRow, int size)
    {
      acc_update_device(_dataPointers[iRow],size*sizeof(T));
    }

    void updateHost(int iRow, int size)
    {
      acc_update_self(_dataPointers[iRow],size*sizeof(T));
    }
}
```

Updating the entire linearized array or a portion of the array is now trivial.

Objects and Functions in OpenACC Loops

Problem

- ➔ FINE/Open is an object oriented C++ code.
- ➔ The loops where parallelism can be exposed are not always at the end of the end of the call tree.
- ➔ OpenACC must support nested function or method calls.

Solution

- ➔ Annotate with **#pragma acc routine seq**.
- ➔ Manage data transfer with dedicated methods.

```
#pragma acc routine seq
double interpolate(double value)

void createDevice()
{
  acc_copyin(this,sizeof(*this));
  acc_create(_data,_dataSize*sizeof(double));
  acc_attach((void**)&_data);
  acc_update_device(_data,_dataSize*sizeof(double));
}

void deleteDevice()
{
  acc_delete(_data,_dataSize*sizeof(double));
  acc_delete(this,sizeof(*this));
}
```

WARNING!
The order is critical, the "this" pointer must be offloaded to the GPU first.

Objects and Functions in OpenACC Loops

Solution

- ➔ Objects may then be included in **present** clauses for parallel loops.
- ➔ This approach may be extended for nested method and function calls.

```
int nbData = 10;
double* foo = double[nbData];
double* bar = double[nbData];

...

acc_copyin(foo,nbData*sizeof(double));
acc_create(bar,nbData*sizeof(double));
interpolator->createDevice();

#pragma acc parallel loop present(foo,bar,fooInterpolator)
for (int i=0; i<nbData; i++)
{
  bar[i] = interpolator>interpolate(foo[i]);
}

acc_copyout(bar,3*sizeof(double));
acc_delete(foo,3*sizeof(double));
fooInterpolator->deleteDevice();
```

Looks quite simple… too simple

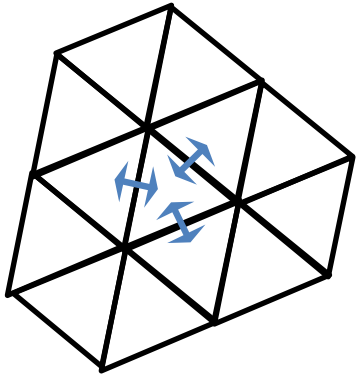
What about more complicated data layouts such as multiple classes containing pointers to data managed elsewhere?

All resolved through careful use of **acc_create()** and **acc_attach().**

Thread Safety

Problem

- ➔ FINE/Open is an unstructured, finite-volume solver.
- ➔ Flux calculations involve a loop over all cell faces, gathering flux contributions in each cell.
- ➔ Multiple faces contribute to each cell, which is not thread safe.



Solution

- ➔ ACC atomic memory operations resolve this with no algorithmic changes.
- ➔ With a maximum of 8 faces per cell collisions are rare, and the performance impact is negligible.

```
#pragma acc parallel loop present(upCell,downCell,flux,…)
for (int iFace=0; iFace<nbFace; iFace++)
{
  int iUpCell   = upCell[iFace];
  int iDownCell = downCell[iFace];
  ...
  < compute flux contributions >
  ...
  #pragma acc atomic
  flux[iUpCell]   -= contribution;
  #pragma acc atomic
  flux[iDownCell] += contribution;
}
```

Example FINE/Open Routine

Viscous Flux Calculation

- ➔ One of many FINE/Open computation "hot spots"
- ➔ Complicated routine requiring all of the techniques described above
  - ◆ Nested calls to templated functions within the targeted loop.
  - ◆ Non-rectangular arrays containing mesh data.
  - ◆ Non-thread safe summation operations.
  - ◆ Access to arrays of flat structs for coordinates

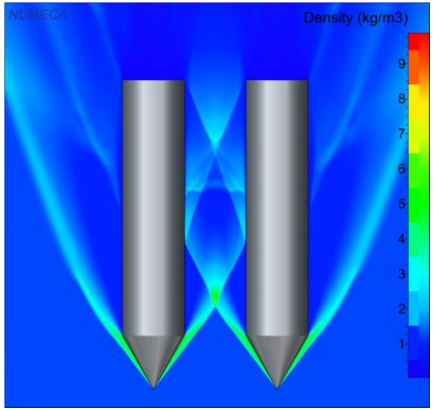
		Viscous Flux Calculation Time (s)		
Mesh Dimension	Number of Cells	Opteron 6284, 1 Core	GPU - NVIDIA K20	GPU Speedup
33 x 33 x 33	32768	0.079	0.011	***7.18***
65 x 65 x 33	131072	0.327	0.034	***9.62***
65 x 65 x 65	262144	0.674	0.058	***11.62***
129 x 65 x 65	524288	1.463	0.112	***13.06***
129 x 129 x 65	1048576	3.165	0.197	***16.07***

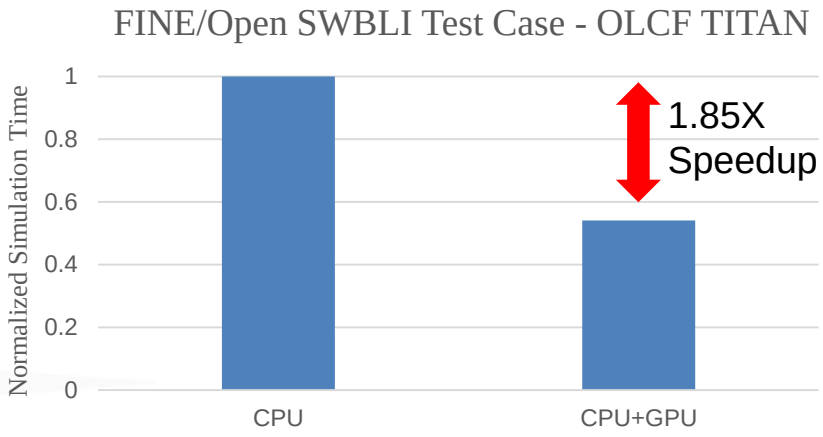
		Viscous Flux Calculation Time (s)		
Mesh Dimension	Number of Cells	Opteron 6284, 8 Cores (MPI)	GPU - NVIDIA K20	GPU Speedup
129 x 129 x 65	1048576	0.491	0.113	***4.35***

Full FINE/Open Performance

Current Status

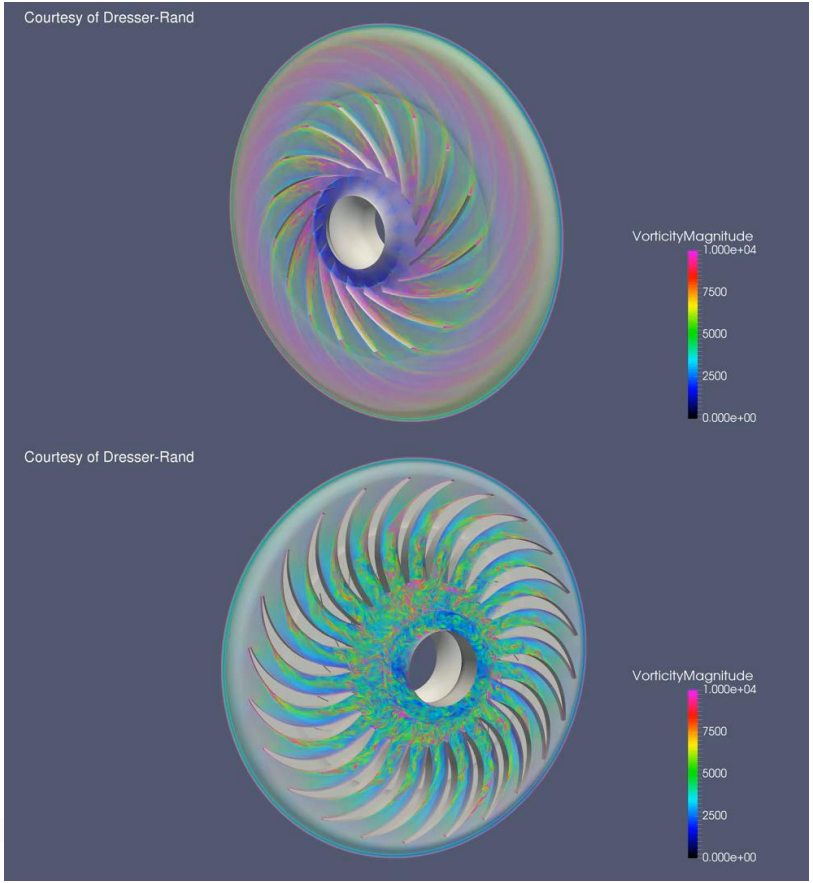
- ➔ Many computation hotspots have been instrumented with OpenACC.
  - Viscous flux calculation.
  - Gradient calculation.
  - Residual smoothing.
  - Artificial dissipation calculation.
  - Thermodynamic quantities lookup and interpolation.
- ➔ High level unstructured data regions minimize transfer of shared or unchanging data.
- ➔ Automatic runtime tuning of the GPU acceleration ensures ideal performance regardless of the target hardware.
- ➔ Approximately 60% of a typical computation time is adapted for heterogeneous execution.
- ➔ The total computation speedup is therefore limited to approximately 2X.





Turbomachinery Rotating Stall Studies

- ➔ Completed in partnership with Dresser-Rand and the Oak Ridge Leadership Computing Facility (OLCF).
- ➔ Large scale, time accurate simulations of multistage turbomachinery configurations passing through stall conditions. (600M cells, 2000+ time steps)
- ➔ All computations completed on the OLCF TITAN Supercomputer, pairing thousands of Opteron CPU cores with hundreds of NVIDIA K20 GPUs per simulation.



Next Steps

FINE/Open

- ➔ The incremental acceleration of the FINE/Open solver will continue, improving the overall simulation speedup and extending support to additional solver modules.
- ➔ Up to 90% of the solver runtime may be eligible for threaded execution, and global speedups of 3X are expected.
- ➔ Support for multiple GPUs is being implemented, ensuring performance on the future OLCF Summit system and other multi-GPU systems.

OpenACC Wish List

- ➔ Improved support and examples for polymorphism (OpenACC 2.6?).
- ➔ Asynchronous threaded execution on both the host and device (concurrent multicore+GPU).
- ➔ ~~A searchable best practices guide with code examples~~ (OpenACC github and textbooks).

NUMECA INTERNATIONAL

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Special Thanks to Dresser Rand

Questions?

Deep Copy of Non-Rectangular Arrays

Example - Initializing a non-rectangular array on a GPU

```
int nbRow   = 5;
AccArray<int> dataAcc;
for (int i=0; i<nbRow; i++) dataAcc.pushRow(i);
dataAcc.allocateData();
dataAcc.createDeviceFast();
int** data = dataAcc.getPointers();

#pragma acc parallel loop present(data)
for (int i=0; i<nbRow; i++)
{
  for (int j=0; j<i; j++) data[i][j] = i;
}

dataAcc.updateHost();
dataAcc.deleteDeviceFast();
std::cout << "Printing data...";
for (int i=0; i<nbRow; i++)
{
  std::cout << "Row " << i << ": " << std::endl;
  for (int j=0; j<i; j++)
  {
    std::cout << data[i][j] << " ";
  }
  std::cout << std::endl;
}
```

```
Printing data...
Row 0:
Row 1: 1
Row 2: 2 2
Row 3: 3 3 3
Row 4: 4 4 4 4
```