# Acceleration of an Adaptive Cartesian Mesh CFD Code in the Current Generation Processor Architectures

Harichand M V[1], Bharatkumar Sharma[2], Sudhakaran G[1], V Ashok[1]

*1 Vikram Sarabhai Space Centre*  *2 Nvidia Graphics*

# Agenda

- What to expect in this presentation?
    - Quick introduction to PARAS3D : CFD code
    - Constraints
    - Learnings

# Quick Background

## ISRO

The Indian Space Research Organization (ISRO) is the primary space agency of the Indian government, and is **among the largest space research organizations in the world**. Its primary objective is to **advance space technology** and use its applications for national benefit, including the development and deployment of communication satellites for television broadcast, telecommunications and meteorological applications, as well as remote sensing satellites for management of natural resources

## VSSC

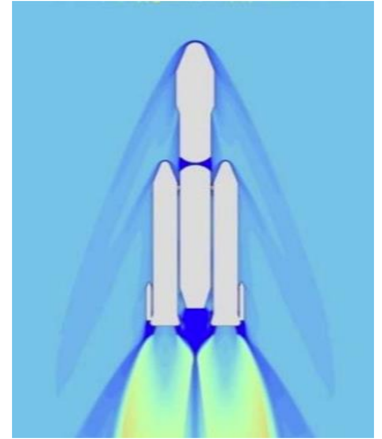
Vikram Sarabhai Space Center is a major space research center of ISRO focusing on rocket and space vehicles for India's Satellite program.

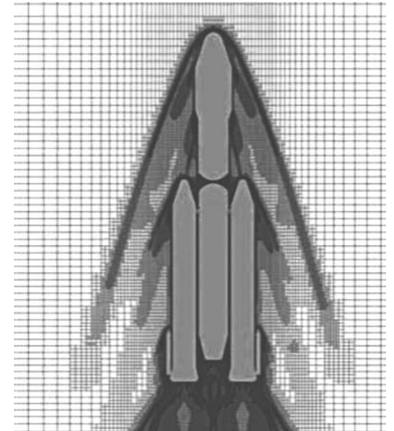
## Supercomputing History

SAGA first supercomputer with GPU in India developed by the Indian Space Research Organization was used to tackle complex aeronautical problems. Listed in Top 500 in June 2012.

# Software Info

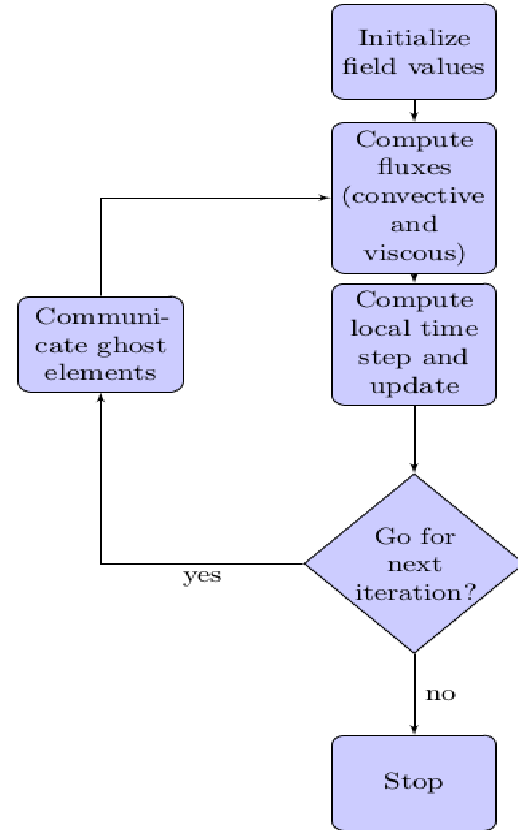
- Used for the aerodynamic design and analysis of launch vehicles in ISRO and aircraft design
- Adaptive Cartesian Mesh Legacy CFD code
- Fully automatic grid generation for any complex geometry
- RANS, Explicit Residual Update, Second Order
- Typical cell count around 50-60 millions





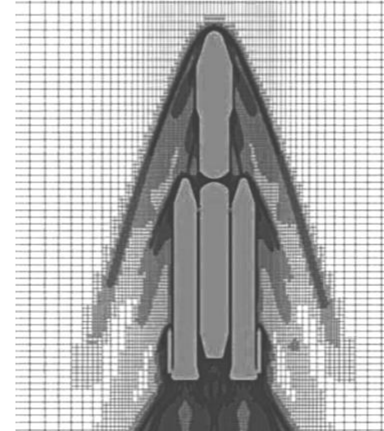
# Solver Flow Chart

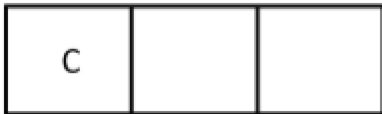
- Compute Fluxes
  - Consists of reconstruction (2nd Order)
  - Riemann Solver for flux computation
  - Requires two level neighbours for each direction
- Compute local time step for each cell
- Update cell value based on fluxes computed
- Explicit update suitable for data parallelism

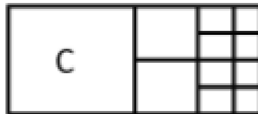


# Mesh Structure.

- Each cell can go 14 levels deep
- Each face, 1 or 4 neighbours
- Cell dependance for reconstruction (Two levels in each direction) on face varies from 2 to  20

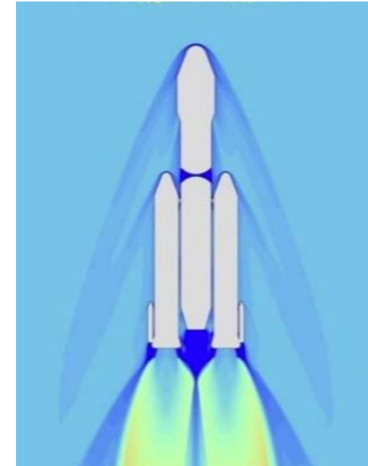




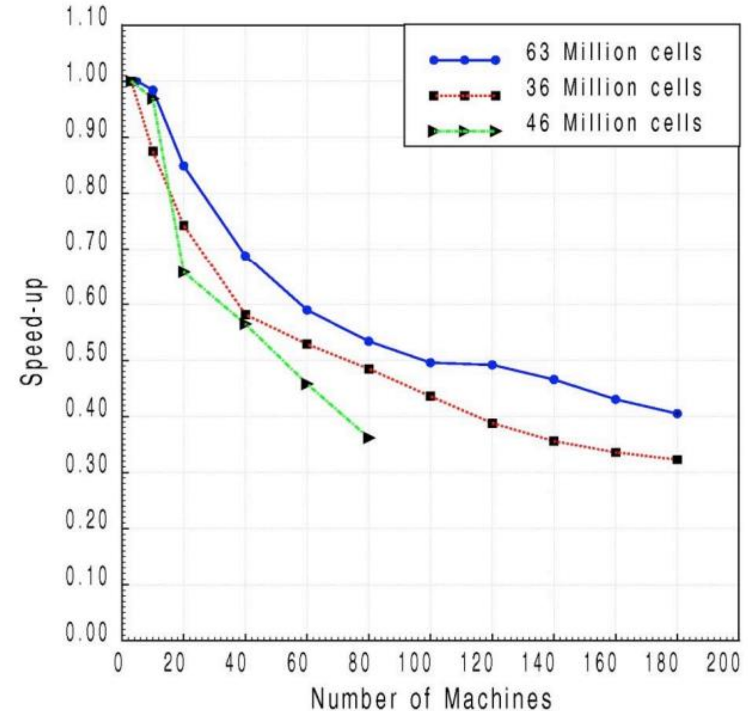
# Features of the legacy solver

- Data structure
  - Forest of oct-trees maintained using child pointers
  - *Lot of pointer chasing while computation*
  - Cell Structure
    - Centroid
    - Size
    - Neighbors (All six directions)
    - Conserved Flow Variable Vector (Internal Energy, Momentum, Density etc.)
  - Face Structure
    - Left and right Cell index
    - Area of the face
    - Axis along which face is aligned (X / Y / Z)
    - Reconstructed variables from left side and right side



# Features of the legacy solver

- MPI Parallelism
  - Each sub-domain is a rectangular box of base cells
  - Synchronous communication of ghost cells
- CUDA C implementation to target GPU
  - 4.5-7x single GPU node vs single CPU node having 2 Quad Core Xeon processors.
  - The speed up depends on the complexity of the geometry, level of grid adaptation and size of the problem under consideration.



# Requirements of New Software

- Should work in hybrid cluster environment
- Easily extensible
  - Maintaining 2 software stack would not be an ideal condition unless required so
- Easy to validate during testing phase
- Ideally adopt to new architecture without fundamental change in code design

# 3 Ways to Accelerate Applications

Applications

Libraries

Compiler Directives

Programming Languages

# OpenACC

**Incremental**

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Enhance Sequential Code

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

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

Begin with a working sequential code.

Parallelize it with OpenACC.

Rerun the code to verify correctness and performance

# OpenACC

**Supported Platforms**

POWER

Sunway

x86 CPU

AMD GPU

NVIDIA GPU

PEZY-SC

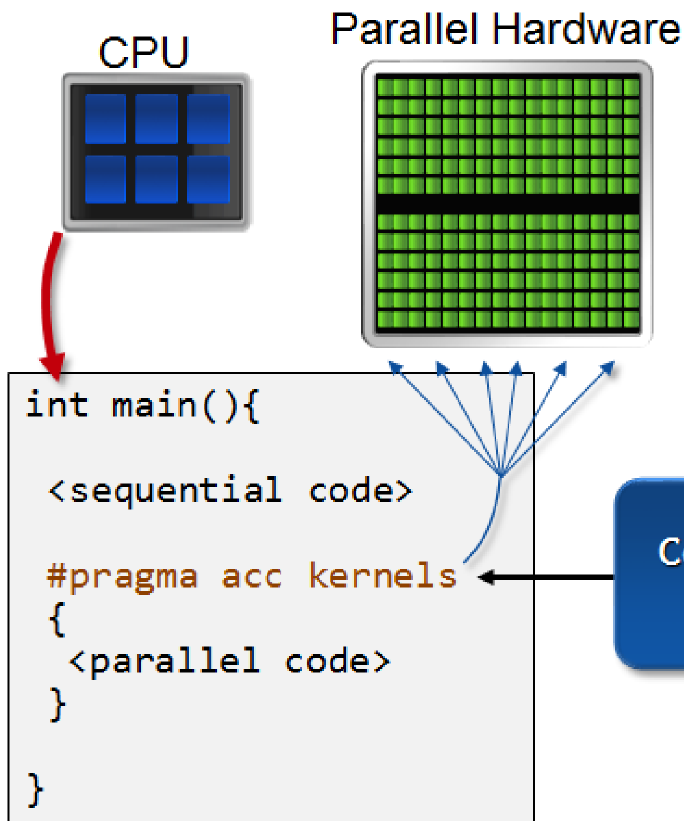
**Single Source**

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

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```
int main(){

...

  #pragma acc parallel loop
  for(int i = 0; i < N; i++)
    < loop code >

}
```

# OpenACC




Compiler Hint

The programmer will give hints to the compiler.

The compiler parallelizes the code.

**Low Learning Curve**

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

# OpenACC

### Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

### Single Source

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

### Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

# 3 Ways to Accelerate Applications

Applications

Libraries

Compiler Directives

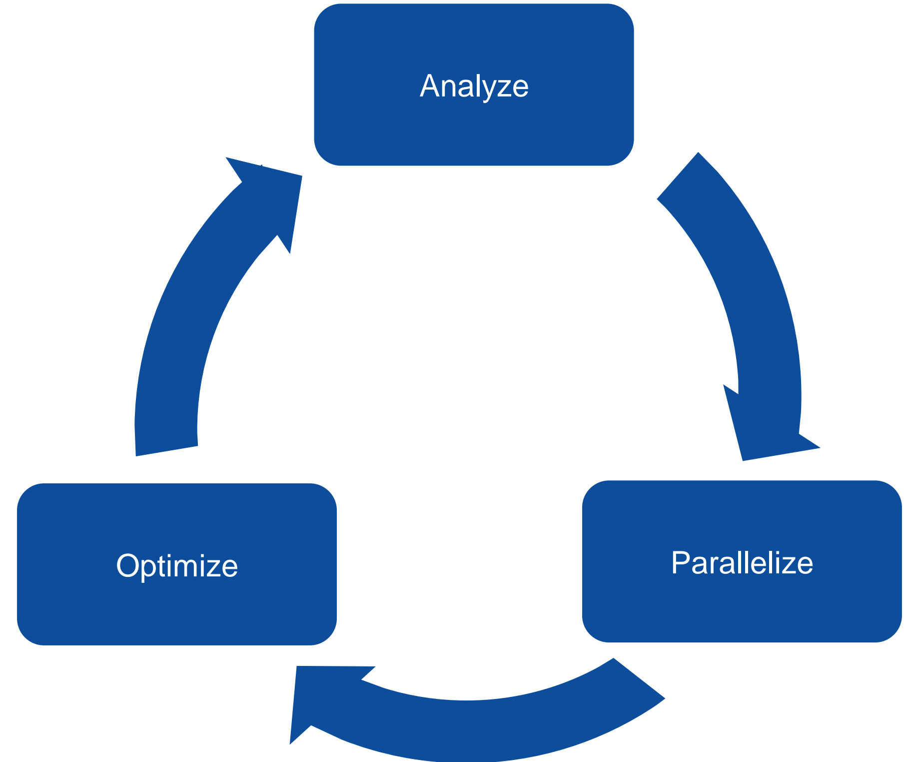
Programming Languages

New version

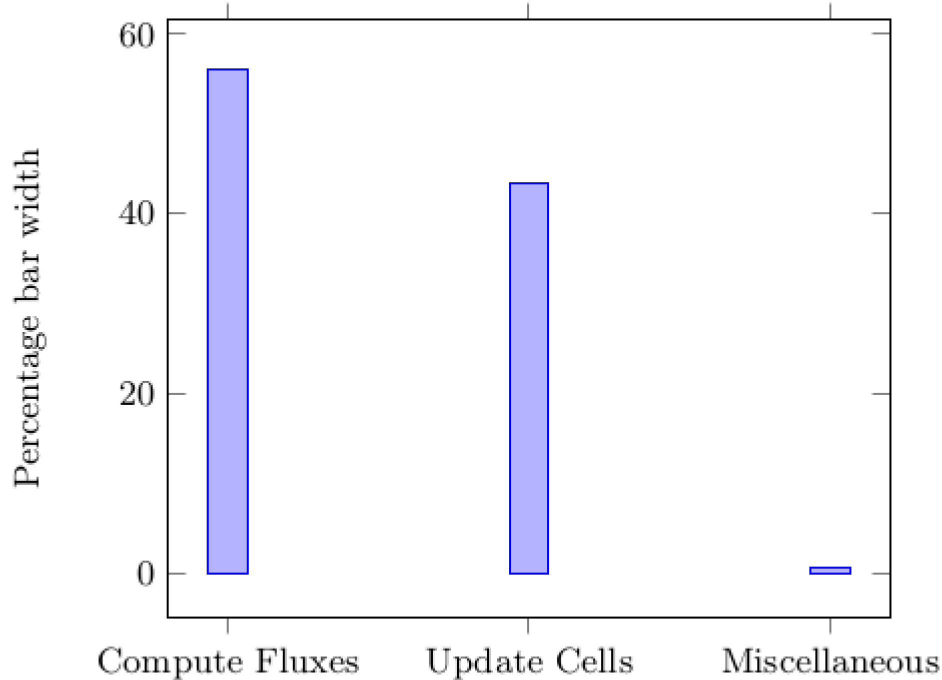
Previous version

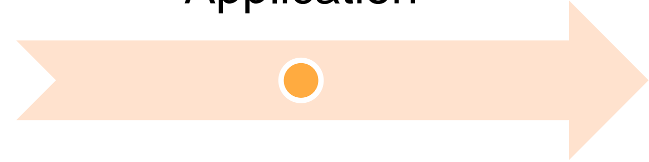
# Development Cycle

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.



# Results (Profiling)



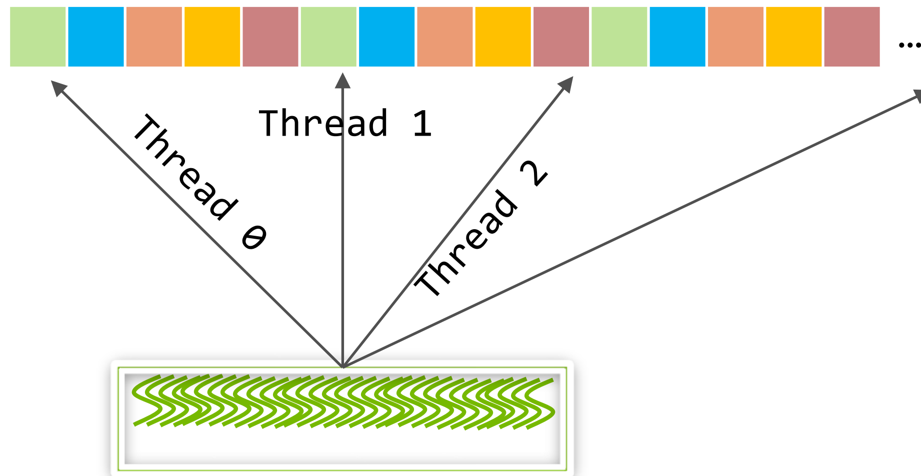




**Observations in Data Layout:**

**layout**:

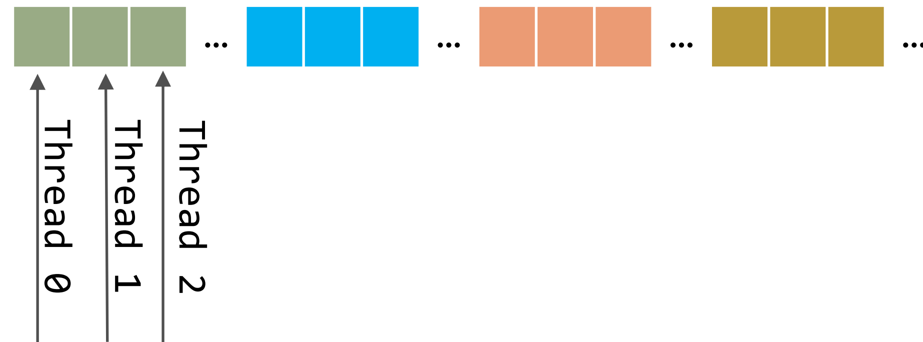
- AOS
- Pointer Chasing because of Oct Tree Structure
- If – Else Statements

```
#define SIZE 1024 * 1024
struct Image_AOS {
        double r;
        double g;
        double b;
        double hue;
        double saturation;
};
Image_AOS gridData[SIZE];
```




**ARRAY OF STRUCTURES**

```
#define SIZE 1024 *1024
struct Image_SOA {
	double r[SIZE];
	double g[SIZE];
	double b[SIZE];
	double hue[SIZE];
	double saturation[SIZE];
};
Image_SOA gridData;
```



```
double u0 = gridData.r[threadIdx.x];
```

**STRUCTURES OF ARRAYS**

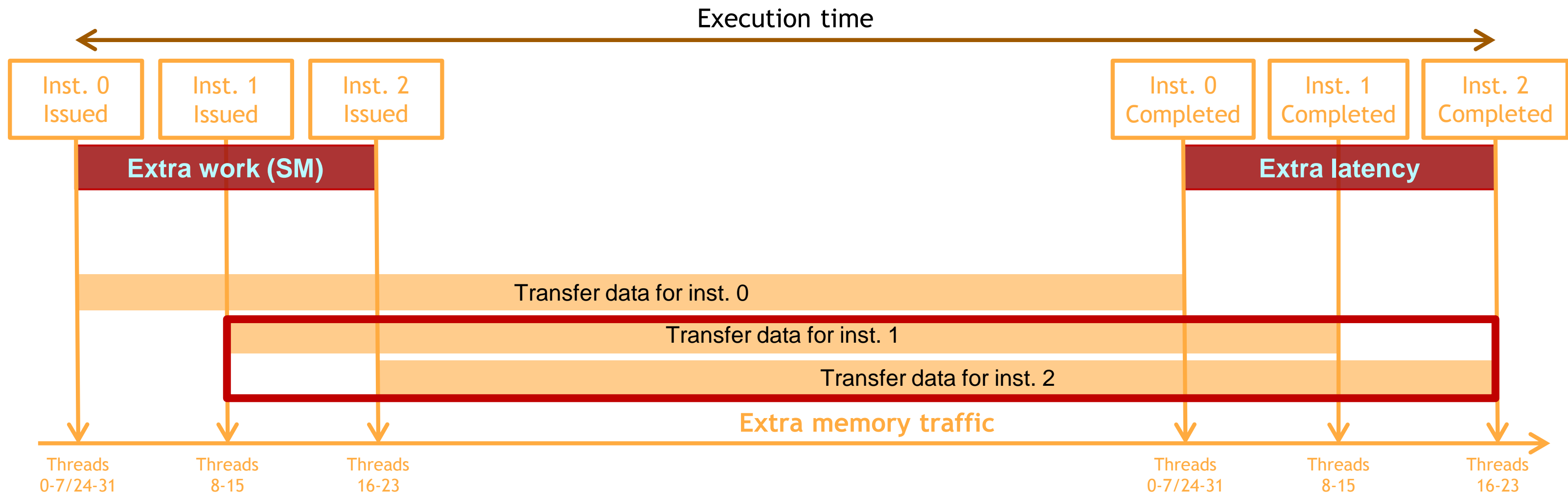
# TRANSACTIONS AND REPLAYS

With replays, requests take more time and use more resources

More instructions issued

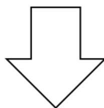
More memory traffic

Increased execution time



# Data Layout

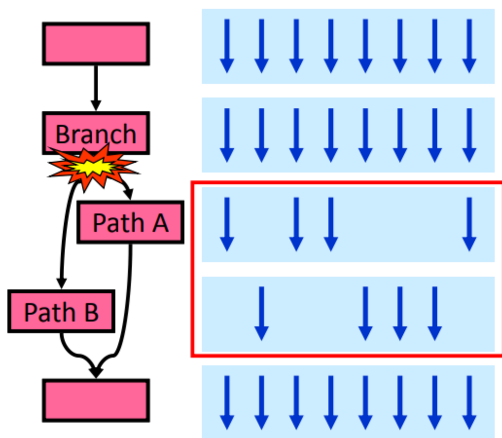
- Structure of Array with in-lined member access

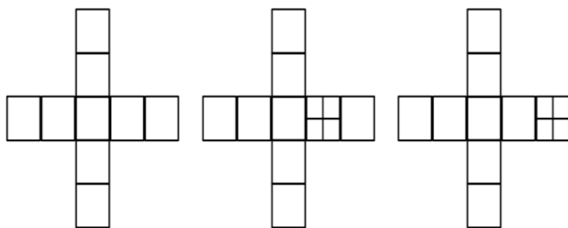


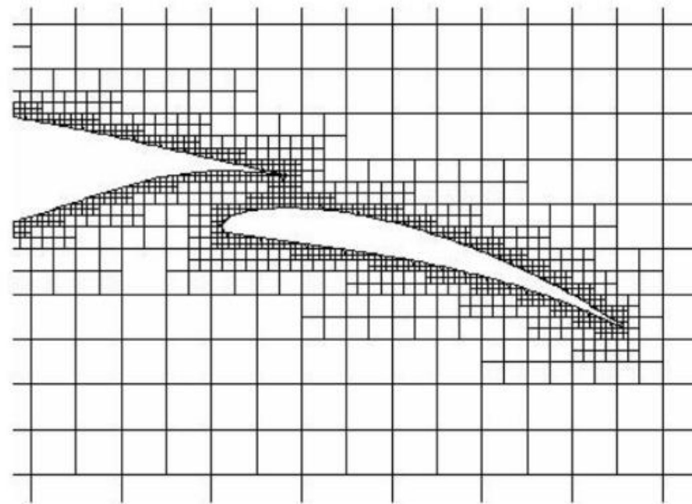
```
struct
{
   int32_t *arr_element_index0;
   int32_t *arr_element_index1;
   real64_t *arr_area0, *arr_area1;
   int8_t *arr_axis, *arr_dir;
   real64_t *arr_fi0r;
   real64_t *arr_fi1r;
}faces_info;


inline int32_t get_face_element_index0(int32_t
      face_index)
{
   return faces_info.arr_element_index0[face_index];
}

inline void set_face_element_index0(int32_t face_index,
      int32_t element_index0)
{
   faces_info.arr_element_index0[face_index] =
         element_index0;
}
```

# Cell Grouping for Data parallelism (GPU Specific)

- Grouped cells into multiple categories based on their data dependency
- Seperate kernel for each group





# CPU Scalability

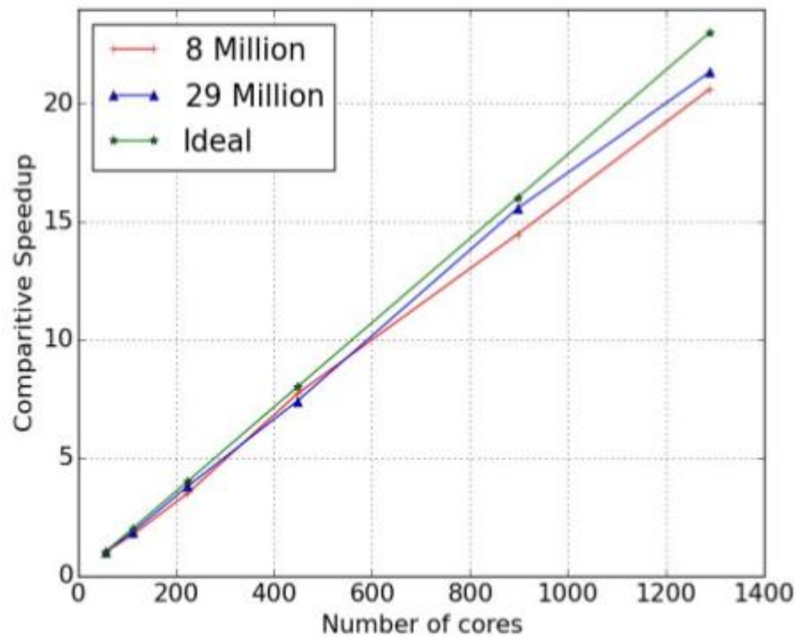
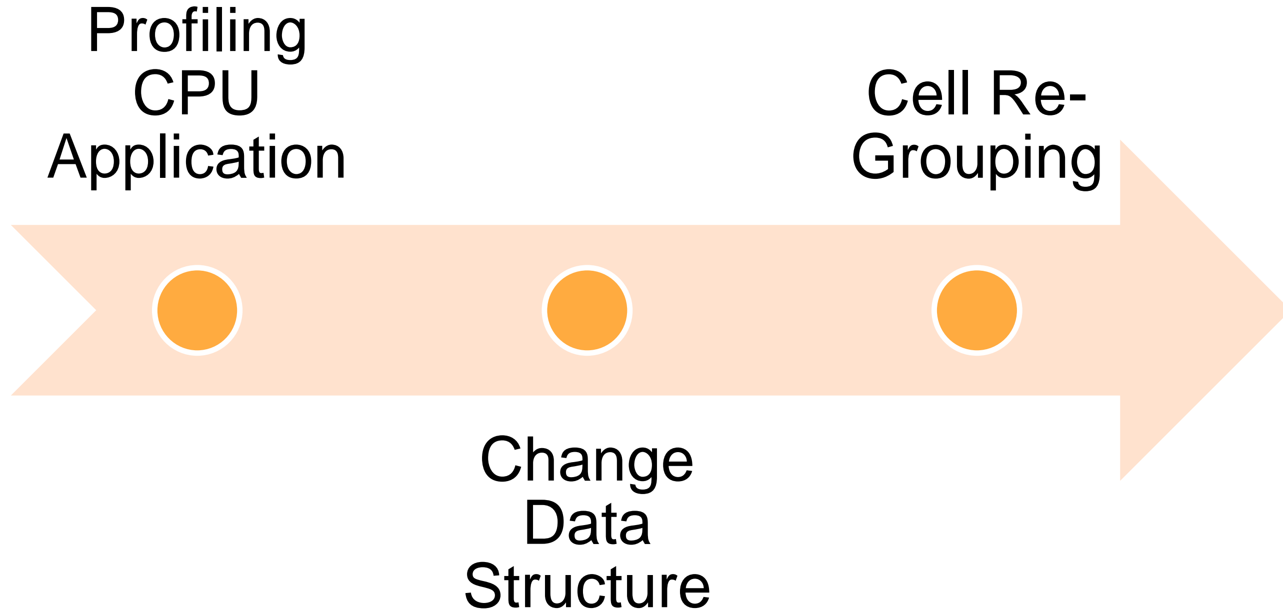
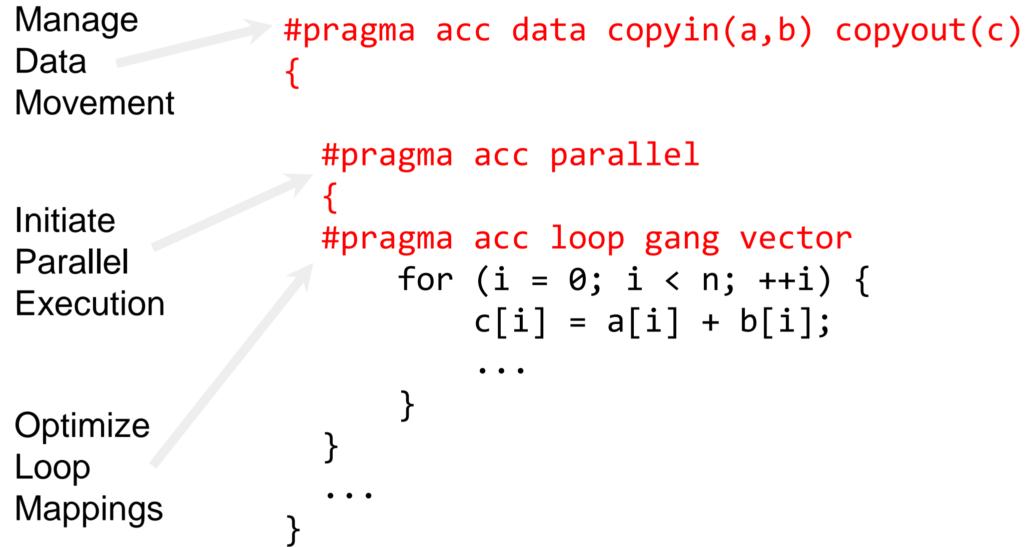



Figure 14: PARAS-3D Scaleup Graph


Profiling CPU Application
Change Data Structure
Cell Re-Grouping

# OpenACC Directives



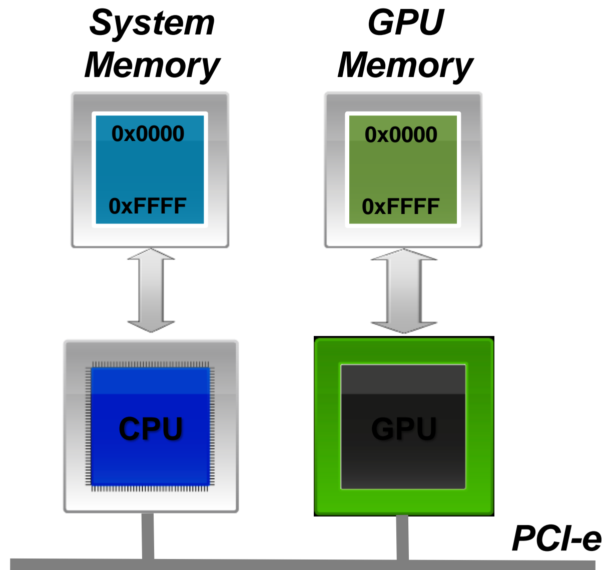
# Parallelize

- Loop parallelism on
  - Reconstruction
  - Flux computation
  - Local time step computation & cell update

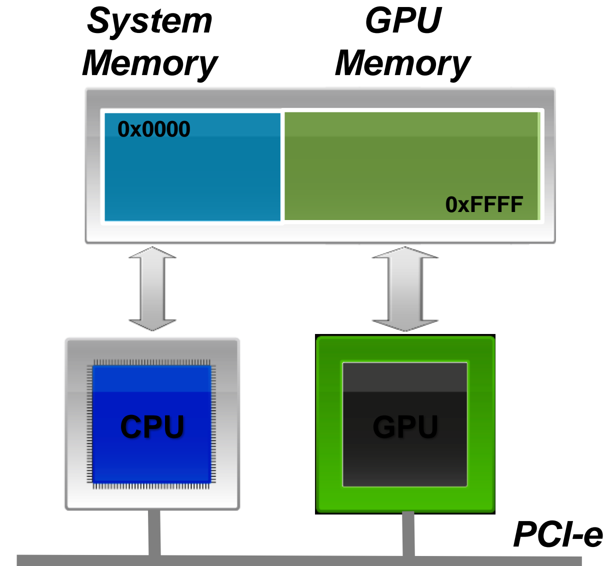
```
#pragma acc parallel loop
for(int32_t i1 = 0; i1 < face_cnt; i1++)
{
   calc_flux(i1);
}

#pragma acc parallel loop
for(int32_t i1 = 0; i1 < (element_count); i1++)
{
   compute_time_step(courant_number, i1);
   update_element_explicit(i1);
}
```

# Unified Virtual Addressing

### *No UVA: Separate Address Spaces*

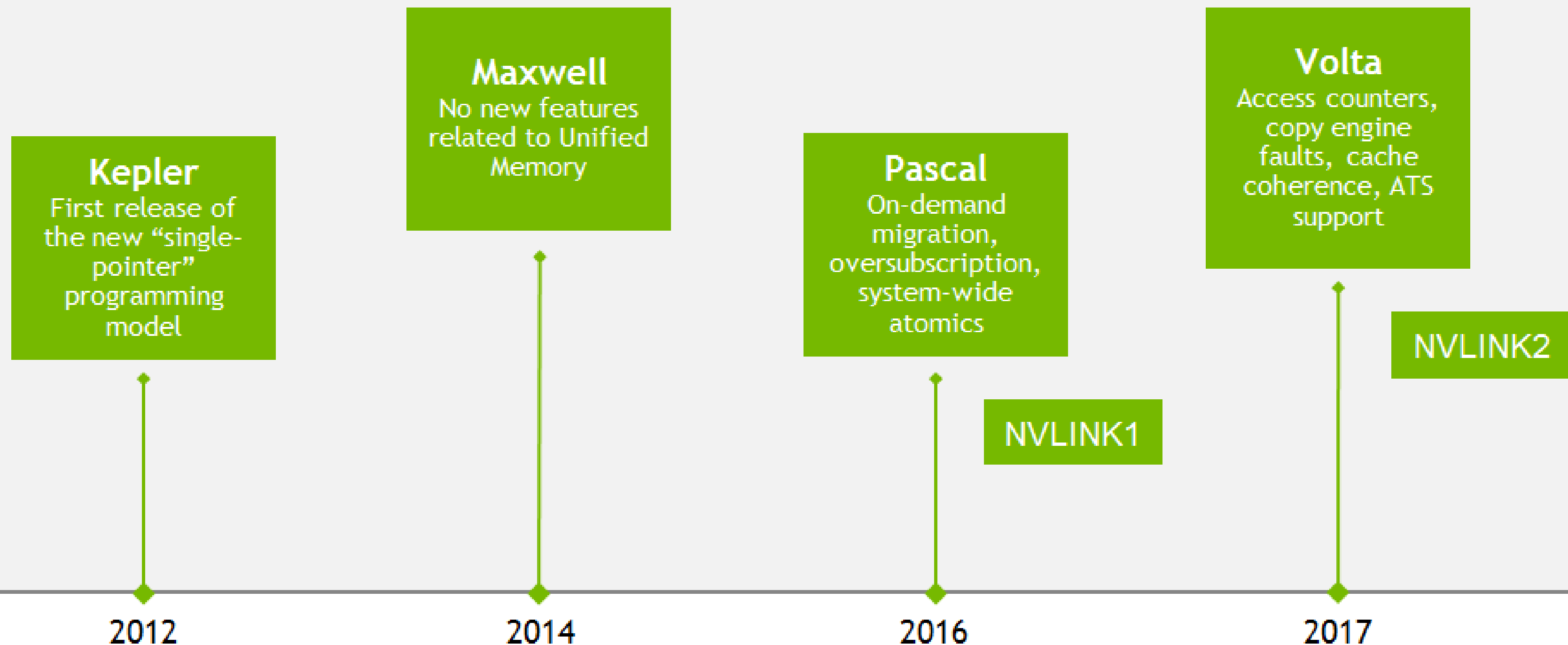


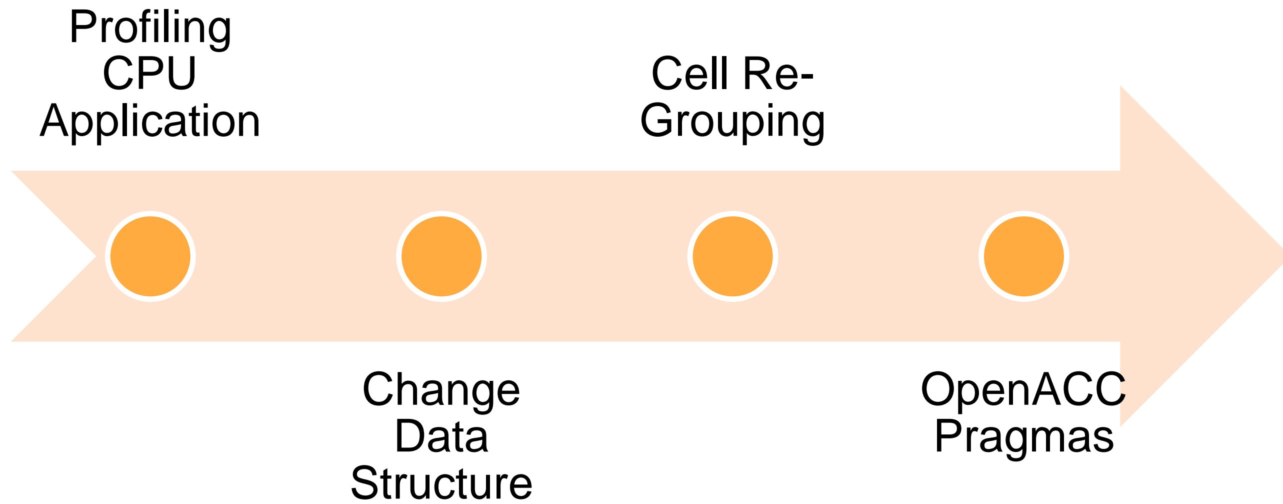

### *UVA: Single Address Space*



# UNIFIED MEMORY

## Evolution of GPU Architectures




Profiling CPU Application
Change Data Structure
Cell Re-Grouping
OpenACC Pragmas

# Results (GPU)

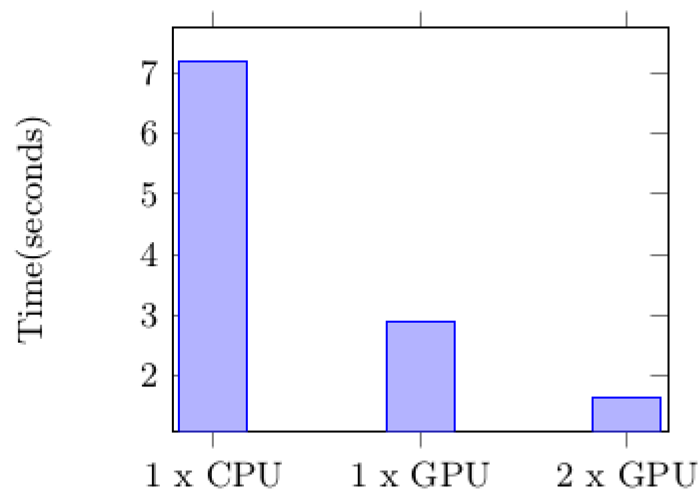



**Figure 16: Comparison of CPU & GPU Performance**

# Analysis

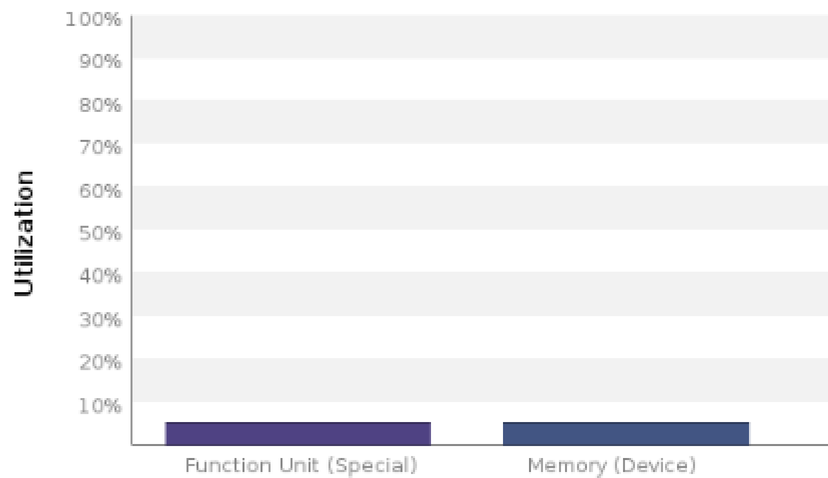
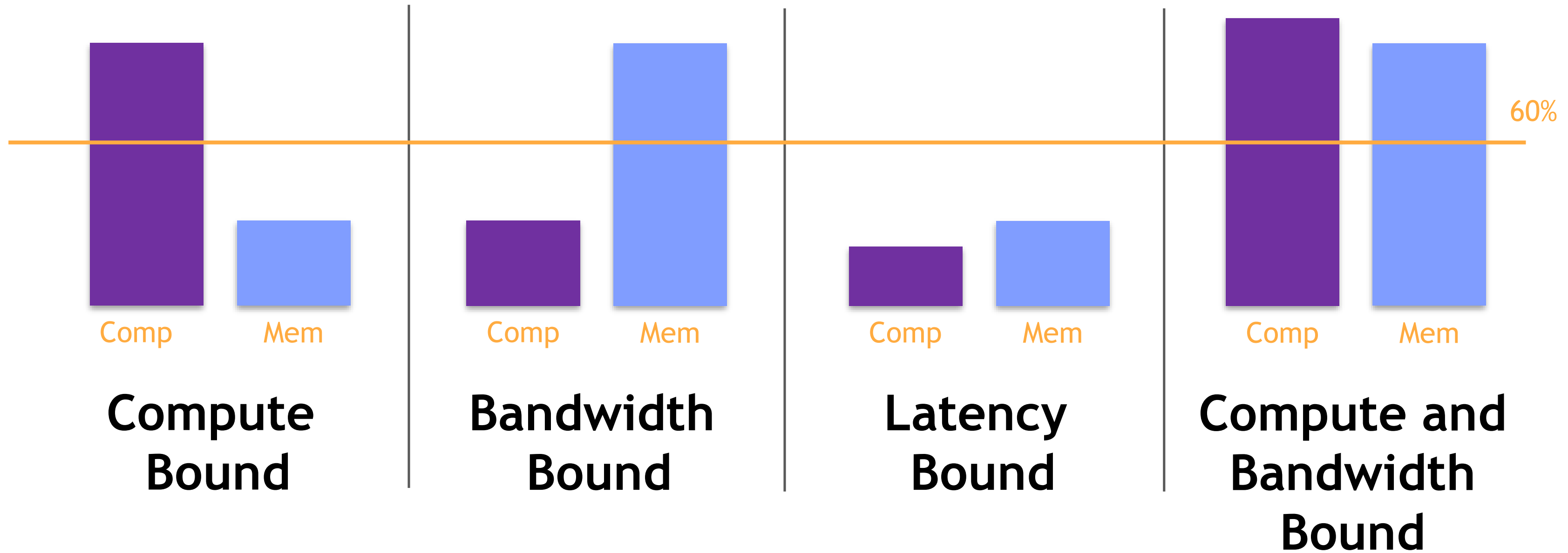



Figure 17: GPU Utilization

# PERFORMANCE LIMITER CATEGORIES

- Memory Utilization vs Compute Utilization
- Four possible combinations:



# DRILL DOWN FURTHER

- Main bottleneck is found to be memory latency
- GPU performance bottle-neck in register spilling and latency
- Kernels used on average 150 Register/Thread

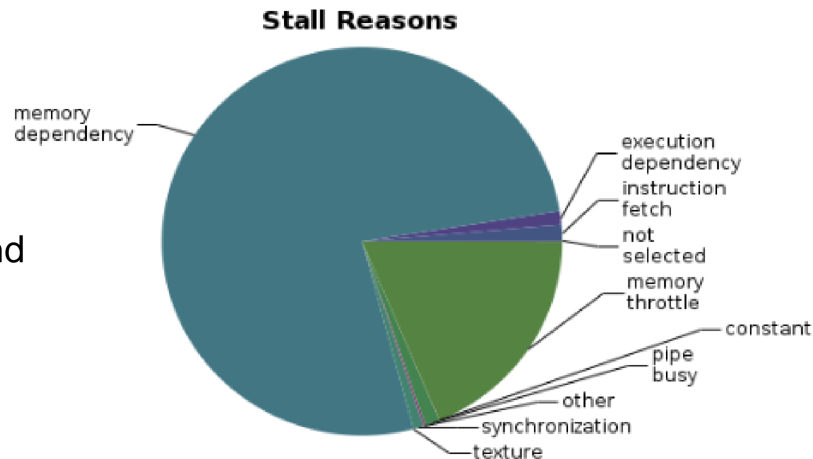



Figure 18: GPU Stall reasons

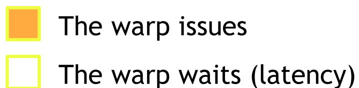
# Occupancy: Know your hardware

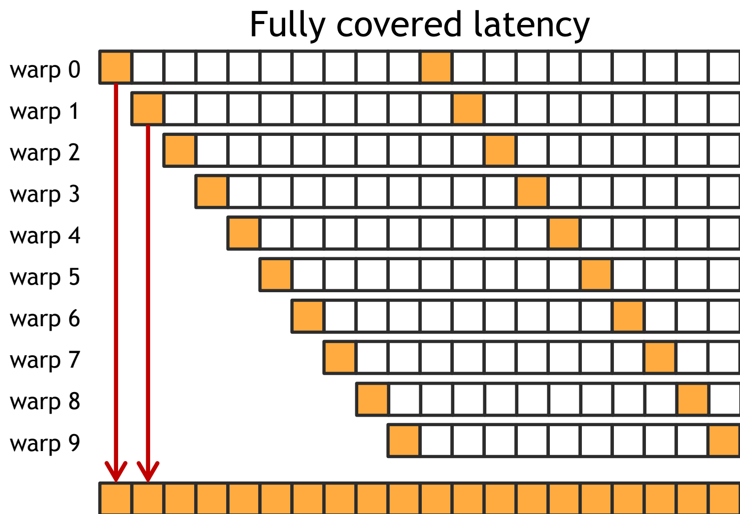
GPU Utilization

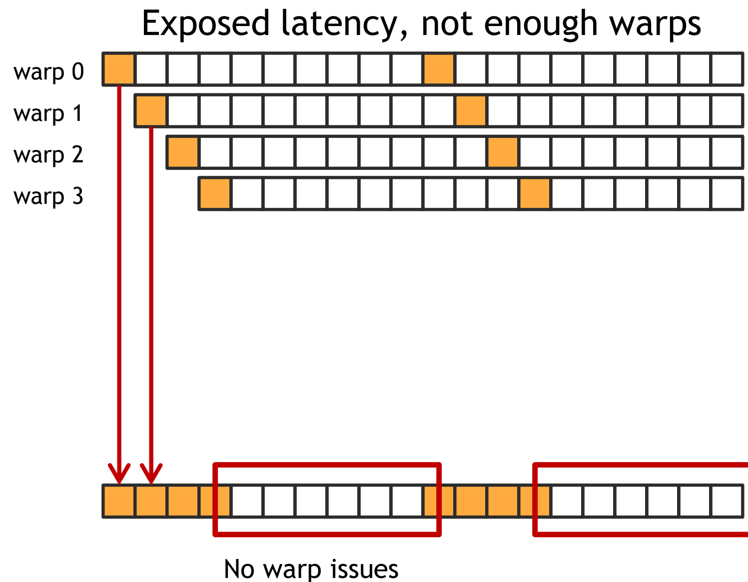
- Each SM has limited resources:
  - max. 64K Registers (32 bit) distributed between threads
  - Max 255 register per thread
  - max. 48KB of shared memory per block (96KB per SMM)
  - Full occupancy: 2048 threads per SM (64 warps)
- When a resource is used up, occupancy is reduced

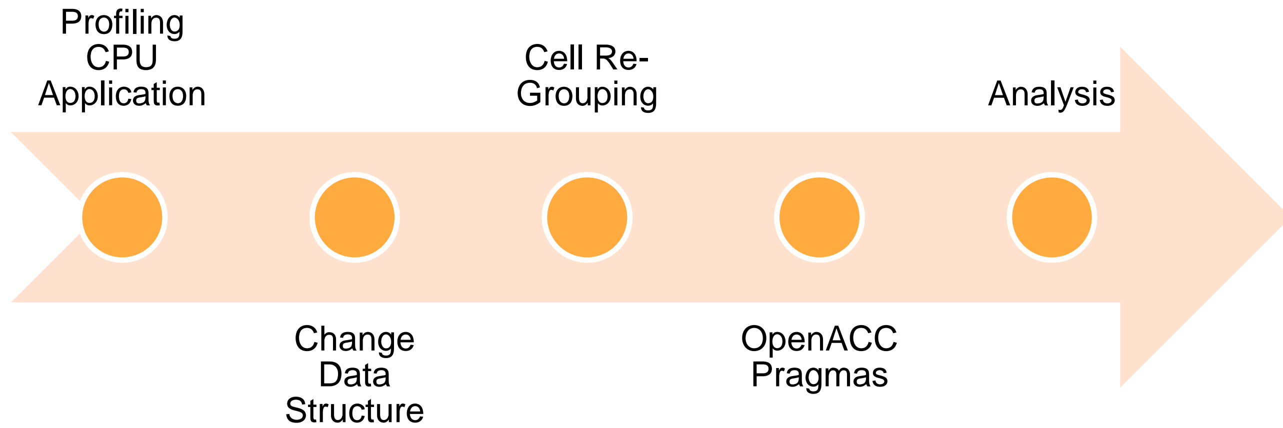
# LATENCY

- GPUs cover latencies by having a lot of work in flight



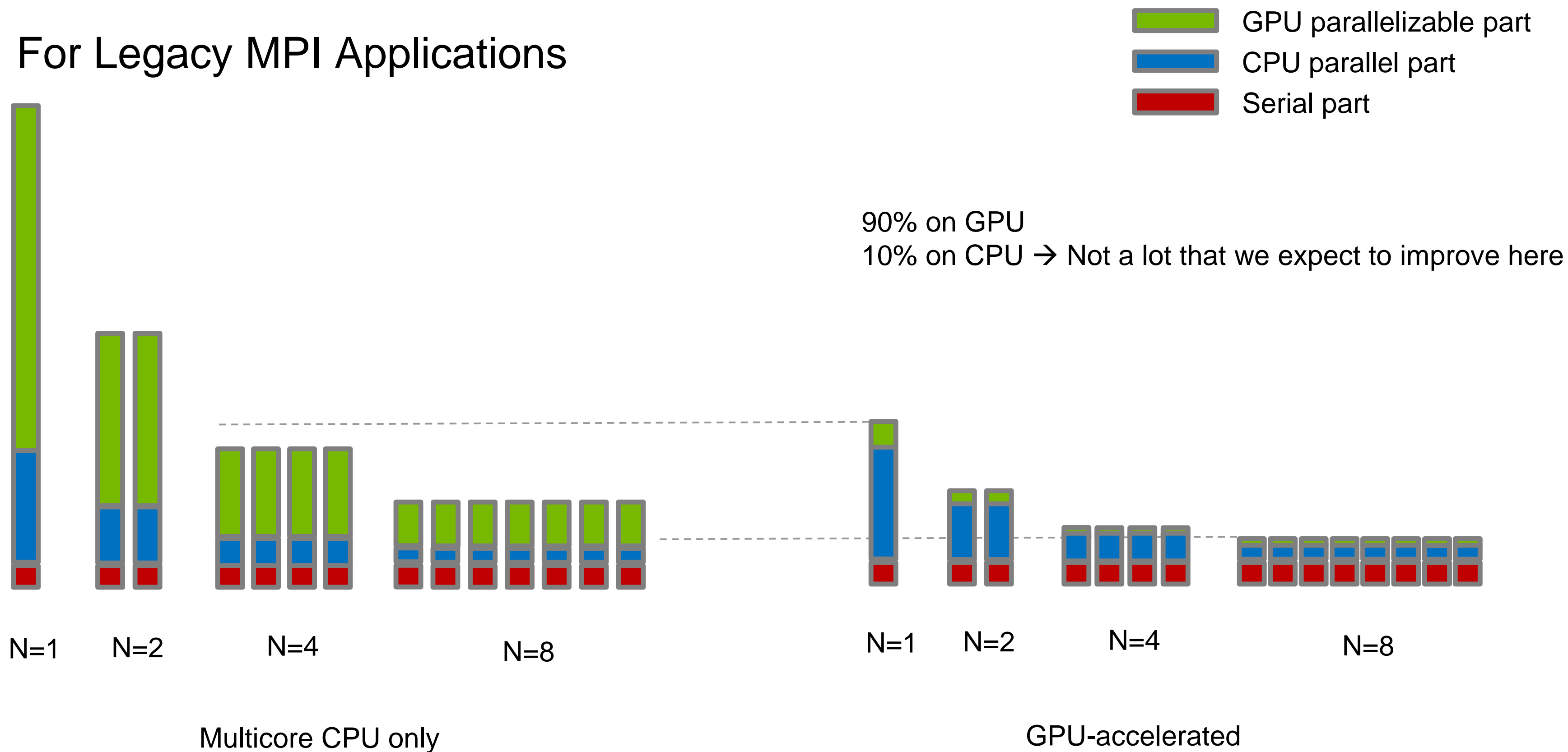

Profiling CPU Application
Change Data Structure
Cell Re-Grouping
OpenACC Pragmas
Analysis

# Optimization Strategies

- Latency Bound: Register Spilling
  - Clean up some unused variable
  - -maxregcount
  - Splitting kernel
- Amdahl’s law
  - MPS

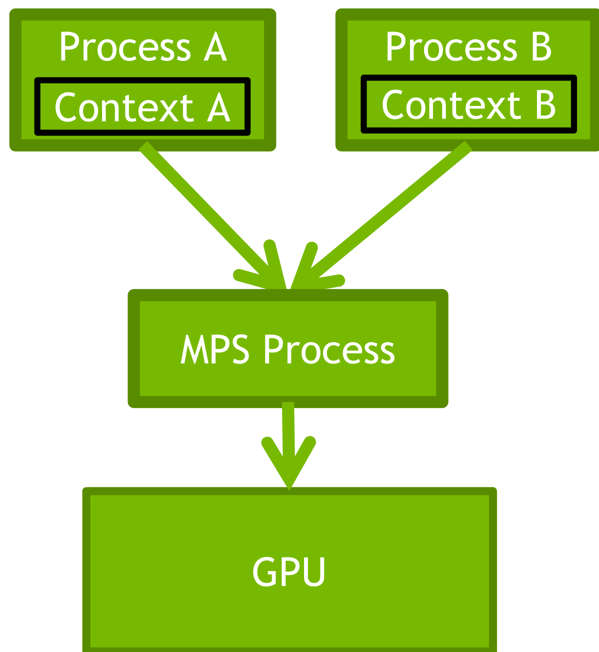
# MULTI Process Service (MPS)

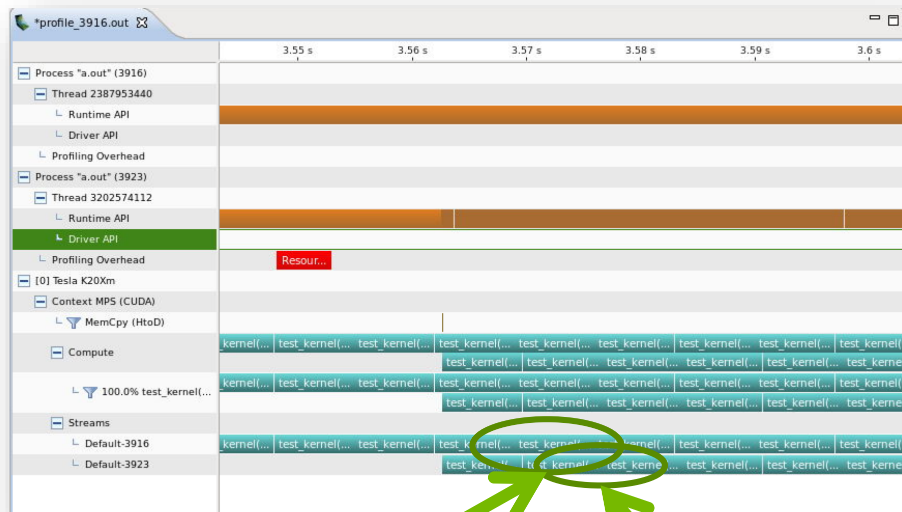
- For Legacy MPI Applications



# Processes sharing GPU with MPS

- Maximum Overlap



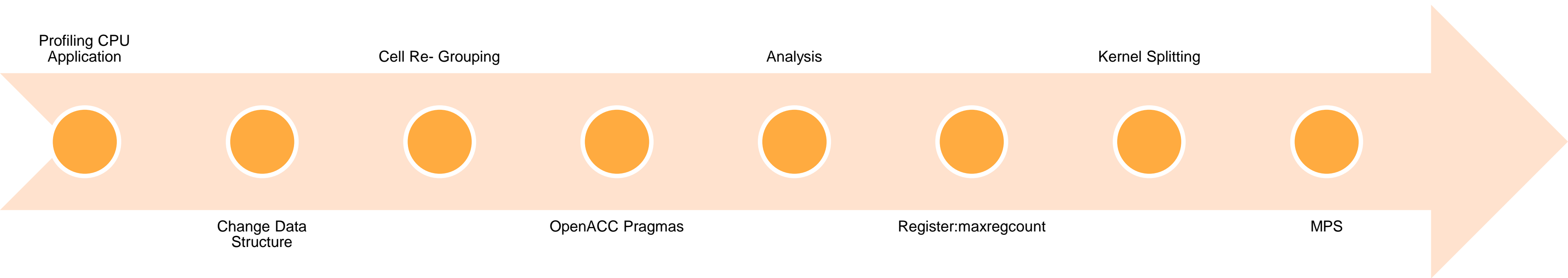




Kernels from Process A

Kernels from Process B

# Results

- 2 X performance grain from the original version ( CPU vs CPU )
- Scalability to thousands of CPU cores
- 4.4 X performance in the Dual Volta GPU version compared to Dual CPU (28 cores Skylake).



Profiling CPU Application
Change Data Structure
Cell Re- Grouping
OpenACC Pragmas
Analysis
Register:maxregcount
Kernel Splitting
MPS

# Conclusion

- A legacy cartesian mesh solver was refactored with 2X performance improvement in CPU
- OpenACC based GPU parallelism improved performance by 4.4 X in Volta GPUs

# Future Work

- Hybrid CPU + GPU computation with asymmetric load partitioning

# Recommendation

- PCAST
    - Helps testing program for correctness, and determine points of divergence.
    - Detecting when results diverge between CPU and GPU versions of code & between the same code run on different processor architectures

```
$ pgcc -Minfo=accel -ta=tesla:autocompare -o a.out example.c

$ PGI_COMPARE=summary,rel=1 ./a.out
comparing a1 in example.c, function main line 26
comparing a2 in example.c, function main line 26
compared 2 blocks, 2000 elements, 8000 bytes
no errors found relative tolerance = 0.100000, rel=1
```

- Unified Memory:
    - Consider using Unified Memory for any new application development.
    - Get your code *running* on the GPU much sooner!