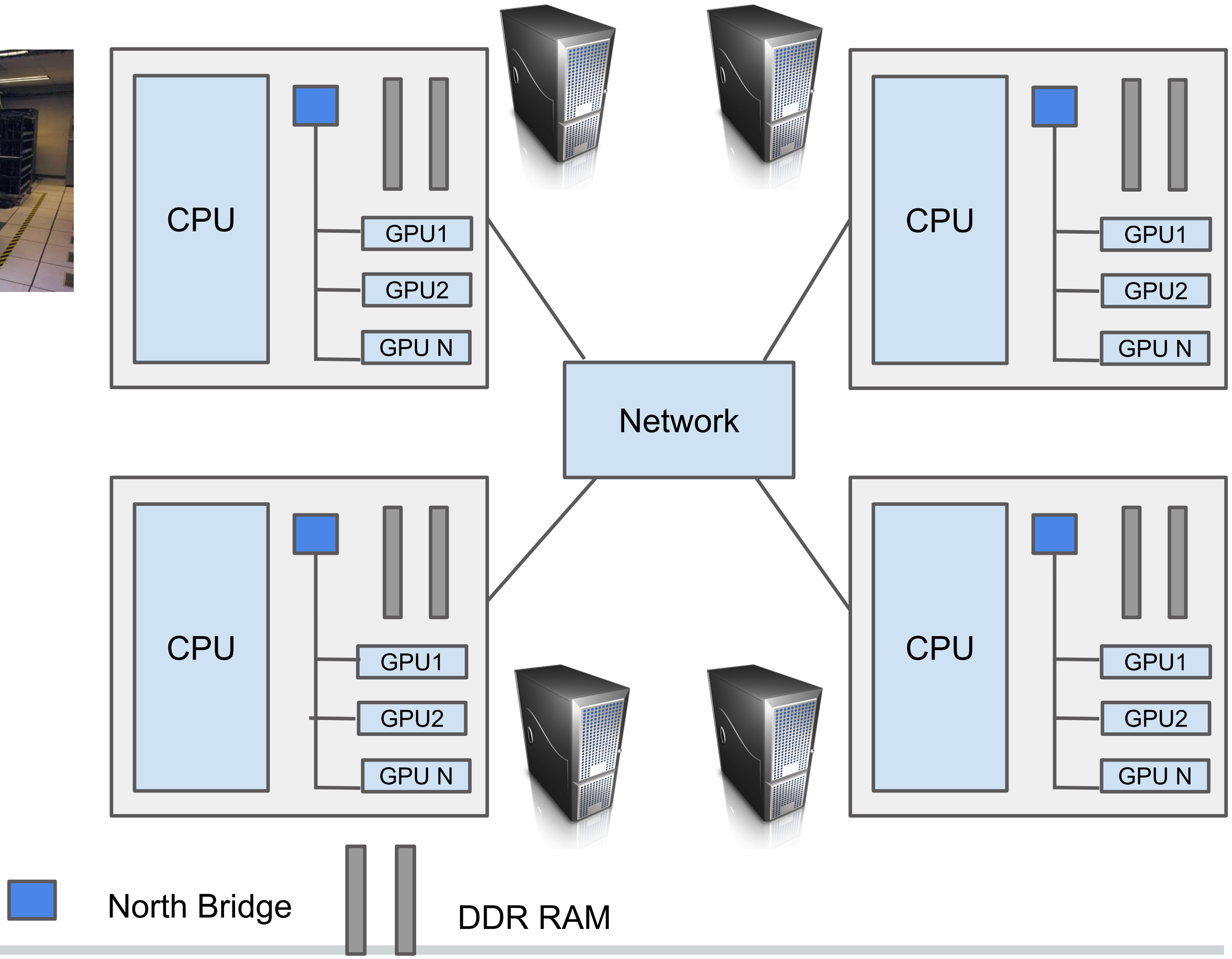
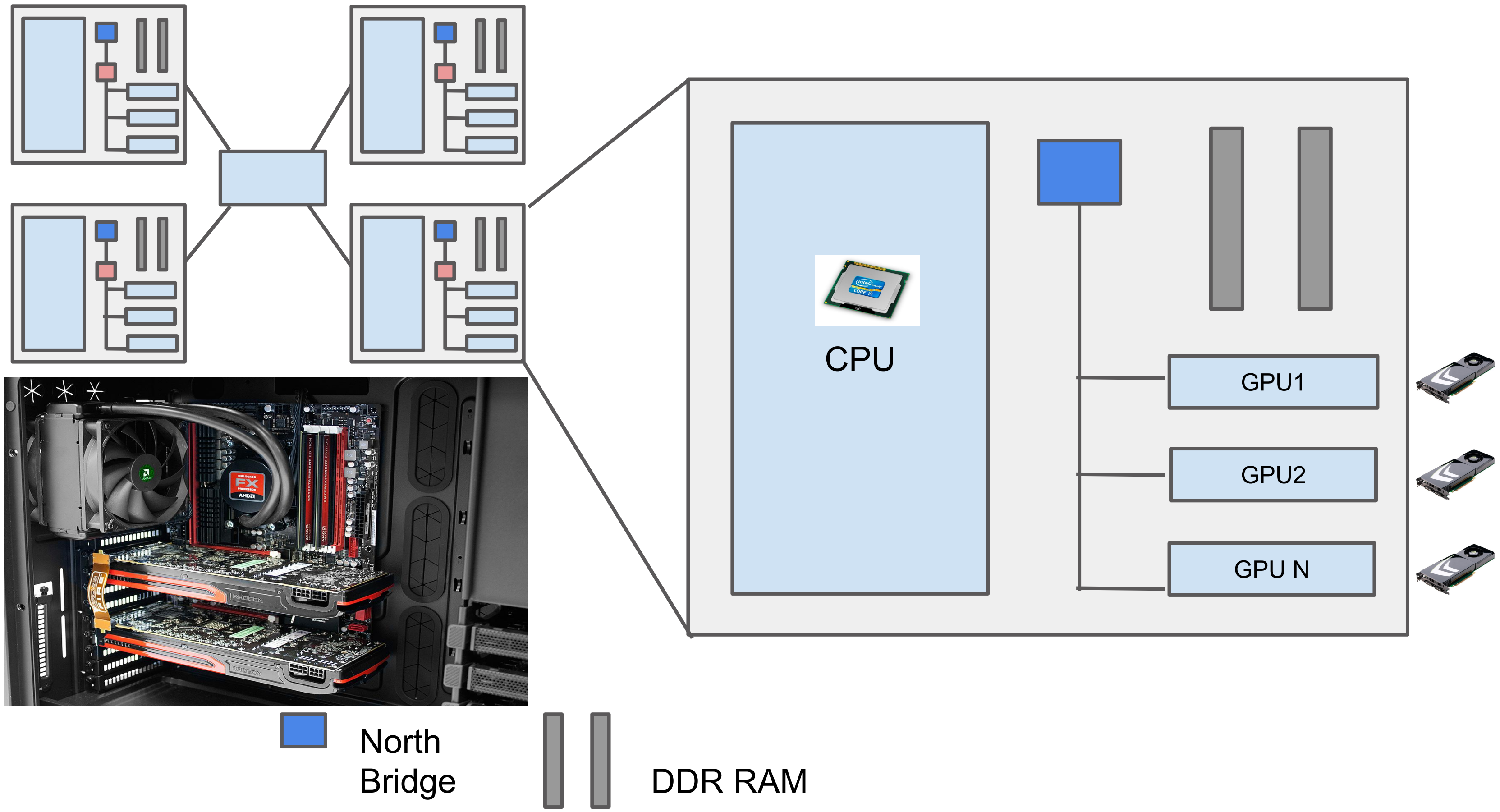
# Automatic Data Allocation, Buffer Management and Data movement for Multi-GPU Machines

Thejas Ramashekar
MSc Engg ( Thesis Defence )
Advisor: Dr. Uday Bondhugula
Indian Institute of Science

# A Typical HPC Setup



# Multi-GPU Machine



## Multi-GPU Setup - Key properties

- Distributed memory architecture

## Multi-GPU Setup - Key properties

- Distributed memory architecture
- Limited GPU memory (512 MB to 6 GB)

## Multi-GPU Setup - Key properties

- Distributed memory architecture
- Limited GPU memory (512 MB to 6 GB)
- Limited PCIex bandwidth (Max 8 GB/s)

## Affine loop nests

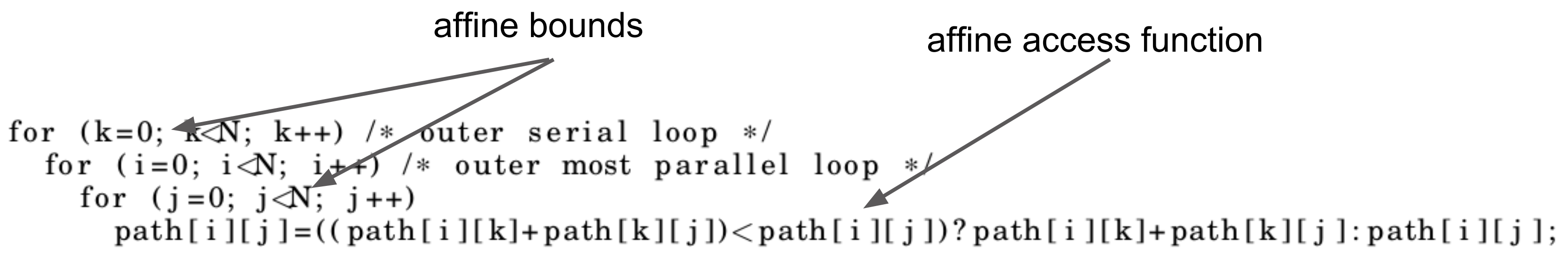
- Loop nests which have affine bounds and the array access functions in the computation statements are affine functions of outer loop iterators and program parameters

## Affine loop nests

- Loop nests which have affine bounds and the array access functions in the computation statements are affine functions of outer loop iterators and program parameters
- eg: stencils, linear-algebra kernels, dynamic programming codes, data mining applications

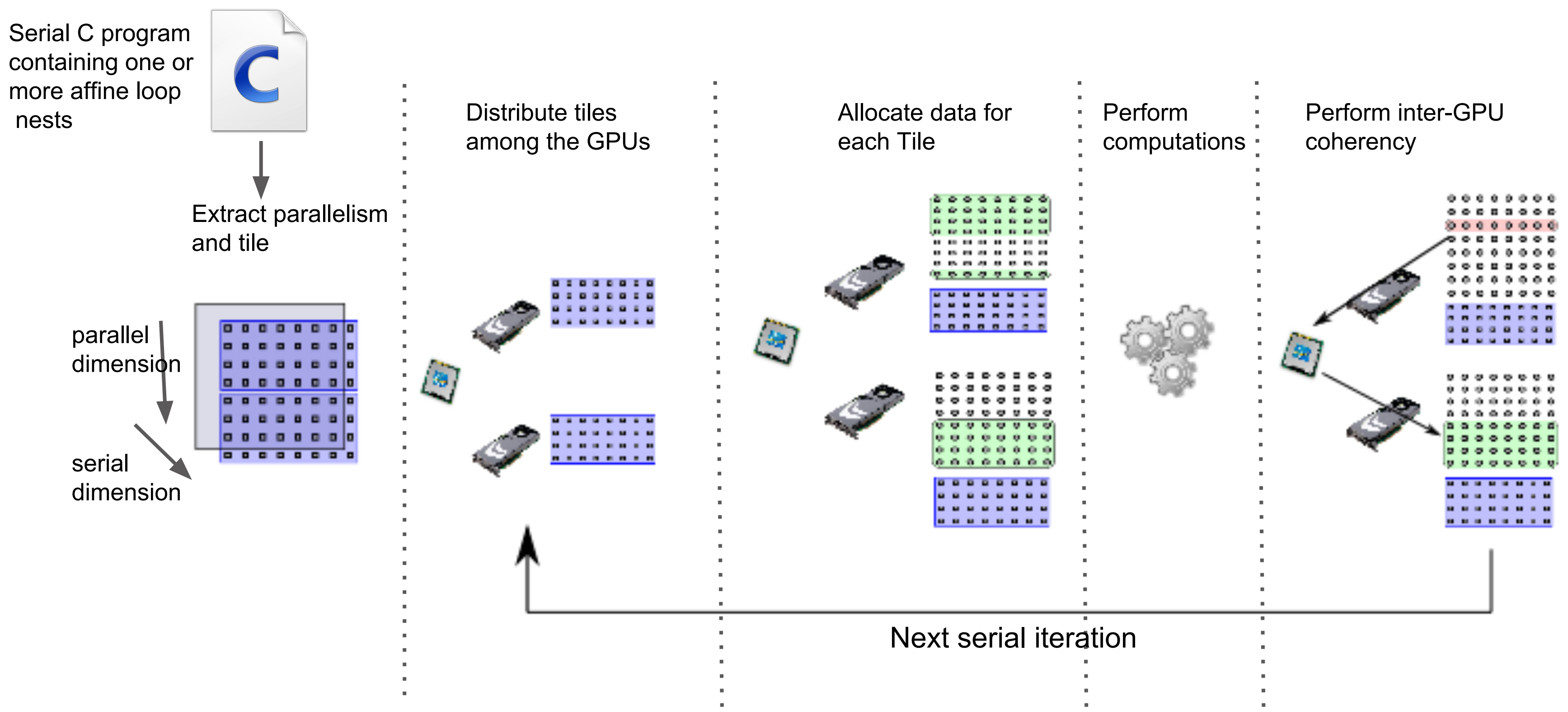
# Affine loop nests

- Loop nests which have affine bounds and the array access functions in the computation statements are affine functions of outer loop iterators and program parameters
- eg: stencils, linear-algebra kernels, dynamic programming codes, data mining applications
- eg: Floyd-Warshall


affine bounds
affine access function

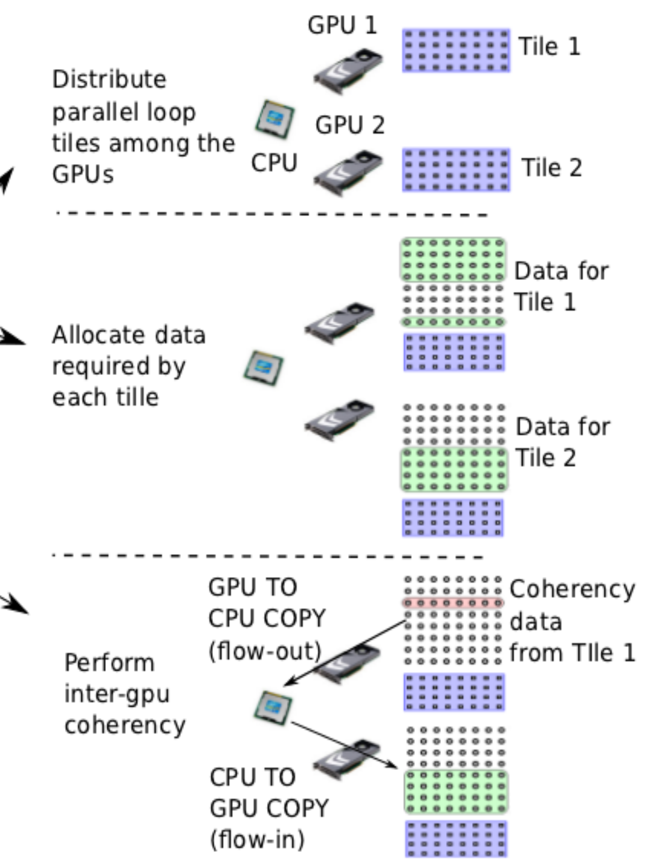
```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```

# Running an affine loop nest on multi-GPU machine


Serial C program containing one or more affine loop nests
Extract parallelism and tile
parallel dimension
serial dimension
Distribute tiles among the GPUs
Allocate data for each Tile
Perform computations
Perform inter-GPU coherency
Next serial iteration

# Structure of an affine loop nest for multi-GPU machine

```
void gpu_mgmt_thread(Device * gpu)
{
 for(tid=gpu−>stile;tid<=gpu−>etile;tid++) {
  allocate_data(gpu, tid);
  launch_kernel(gpu, tid);
  perform_coherency(gpu, tid);
 }
}
int main()
{
 for(ser=0; ser < NUM_ITER; ser++) {
  for(i=0;i<NUM_GPUS;i++) {
   distribute_parallel_loop(i,&gpu[i].stile,&gpu[i].etile);
   spawn_thread(&gpu[i], gpu_mgmt_thread);
  }
  synchronize_mgmt_threads();
 }
 aggregate_results();
 ...
 // more affine loop nests can follow with the same structure
}
```



Distribute parallel loop tiles among the GPUs
CPU
GPU 1
GPU 2
Tile 1
Tile 2
Allocate data required by each tille
Data for Tile 1
Data for Tile 2
Perform inter-gpu coherency
GPU TO CPU COPY (flow-out)
CPU TO GPU COPY (flow-in)
Coherency data from TIle 1

## The need for a multi-GPU memory manager

- Manual programming of multi-GPU systems is tedious, error-prone and time consuming

## The need for a multi-GPU memory manager

- Manual programming of multi-GPU systems is tedious, error-prone and time consuming
- Existing works are either:
  - Manual application specific techniques

    or
  - Have inefficiencies in terms of data allocation sizes, reuse exploitation, inter-GPU coherency etc

## Design goals for a multi-GPU memory manager

- The desired abilities for a multi-GPU memory manager are:

## Design goals for a multi-GPU memory manager

- The desired abilities for a multi-GPU memory manager are:
  - To identify and minimize data allocation sizes

## Design goals for a multi-GPU memory manager

- The desired abilities for a multi-GPU memory manager are:
    - To identify and minimize data allocation sizes
    - To reuse data already present on the GPU

## Design goals for a multi-GPU memory manager

- The desired abilities for a multi-GPU memory manager are:
  - To identify and minimize data allocation sizes
  - To reuse data already present on the GPU
  - To keep data transfers minimal and efficient

## Design goals for a multi-GPU memory manager

- The desired abilities for a multi-GPU memory manager are:
    - To identify and minimize data allocation sizes
    - To reuse data already present on the GPU
    - To keep data transfers minimal and efficient
    - To achieve all the above with minimal overhead

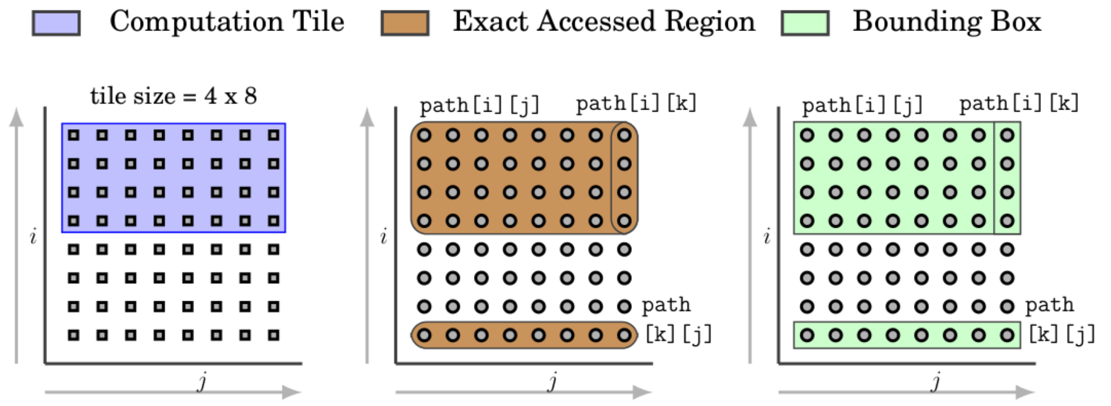
## Bounding Boxes

- Bounding box of an access function, is the smallest hyper-rectangle that encapsulates all the array elements accessed by it

# Bounding Boxes

- Bounding box of an access function, is the smallest hyper-rectangle that encapsulates all the array elements accessed by it

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```



(a) Iteration space of a tile (size: 4 x 8, $k$=7)

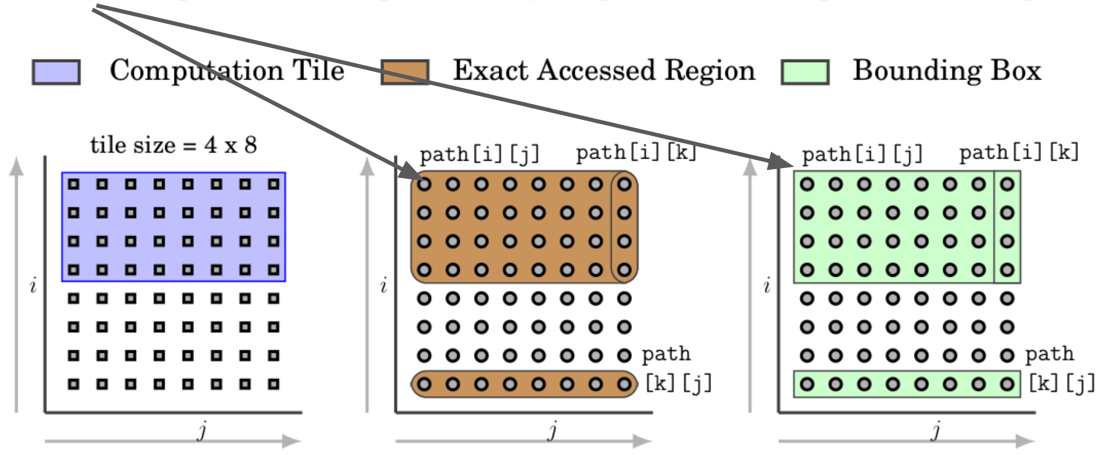
(b) Exact array regions accessed by the tile

(c) bounding boxes

## Bounding Boxes

- Bounding box of an access function, is the smallest hyper-rectangle that encapsulates all the array elements accessed by it

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```

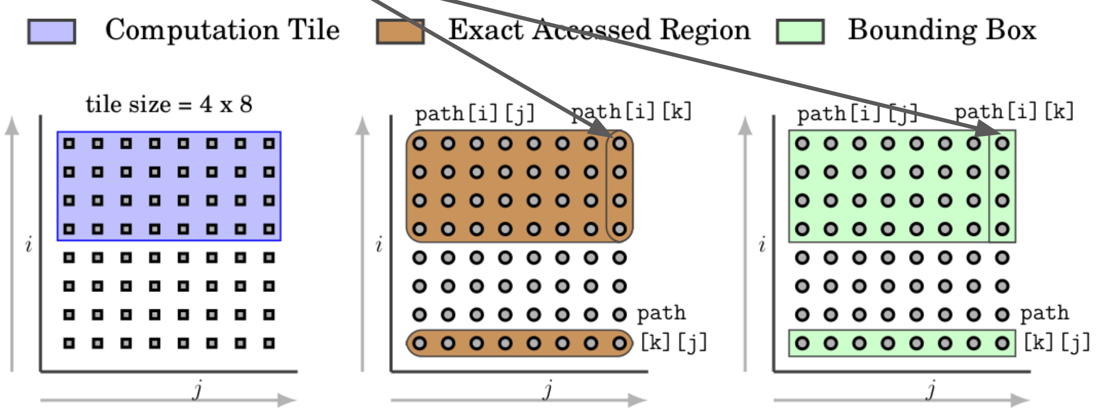


(a) Iteration space of a tile (size: 4 x 8, $k$=7) (b) Exact array regions accessed by the tile (c) bounding boxes

## Bounding Boxes

- Bounding box of an access function, is the smallest hyper-rectangle that encapsulates all the array elements accessed by it

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```



(a) Iteration space of a tile (size: 4 x 8, $k$=7)

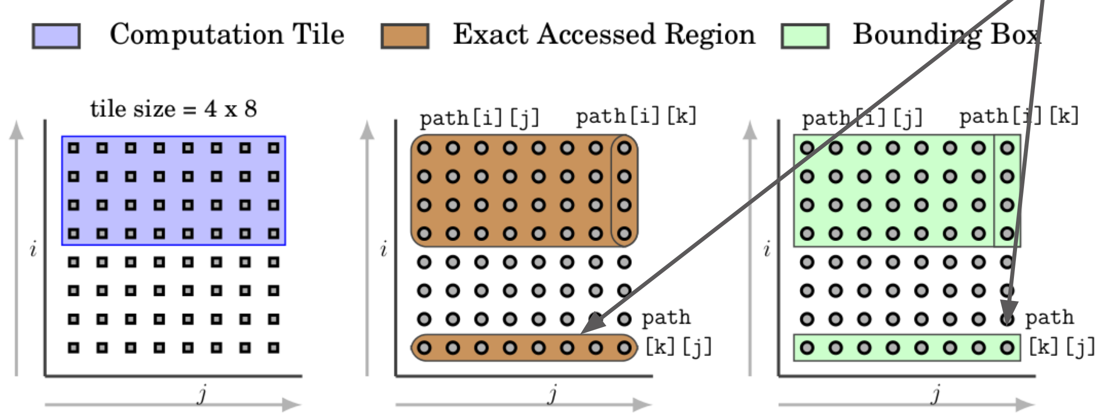
(b) Exact array regions accessed by the tile

(c) bounding boxes

## Bounding Boxes

- Bounding box of an access function, is the smallest hyper-rectangle that encapsulates all the array elements accessed by it

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```



(a) Iteration space of a tile (size: 4 x 8, $k$=7) (b) Exact array regions accessed by the tile (c) bounding boxes

## Key insights on bounding boxes

- Two key insights:

## Key insights on bounding boxes

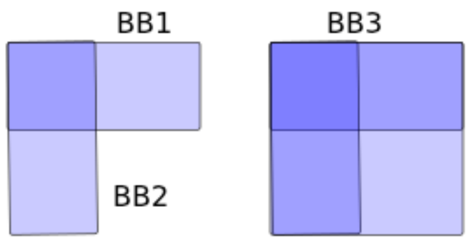
- Two key insights:
    - Bounding boxes can be subjected to standard set operations at runtime with negligible overhead

## Key insights on bounding boxes

- Two key insights:
    - Bounding boxes can be subjected to standard set operations at runtime with negligible overhead
    - GPUs have architectural support for fast rectangular copies

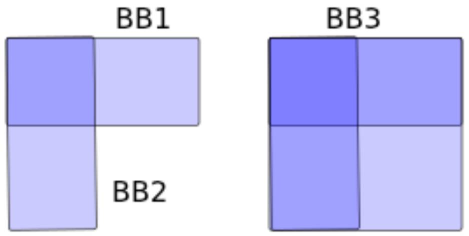
## Set Operations on Bounding Boxes

bb_convex_union(BB1,BB2) = BB3

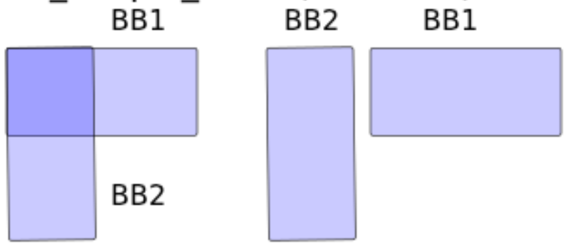


## Set Operations on Bounding Boxes

bb_convex_union(BB1,BB2) = BB3

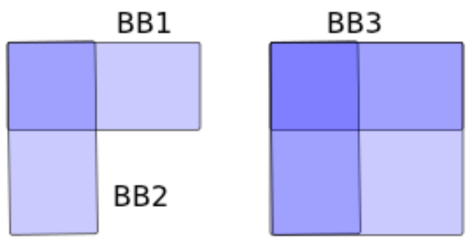


bb_simple_union(BB1,BB2) = BB1 + BB2

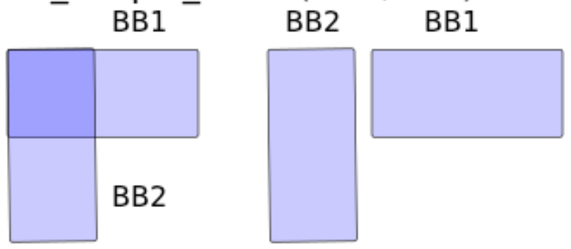


# Set Operations on Bounding Boxes

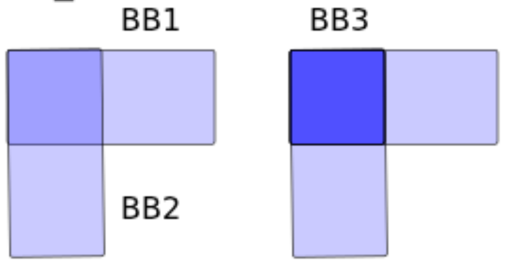
bb_convex_union(BB1,BB2) = BB3



bb_simple_union(BB1,BB2) = BB1 + BB2

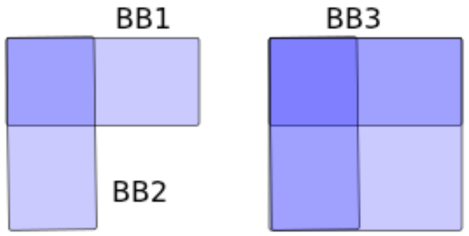


bb_intersection(BB1,BB2) = BB3

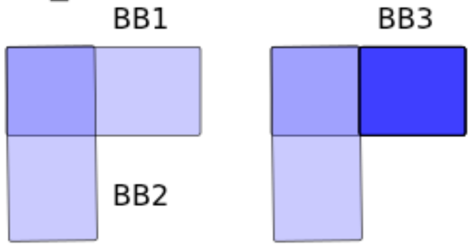


# Set Operations on Bounding Boxes

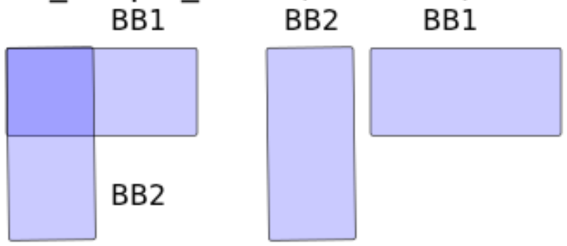
bb_convex_union(BB1,BB2) = BB3



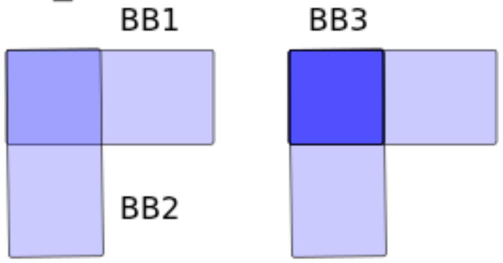
bb_subtract(BB1,BB2) = BB3



bb_simple_union(BB1,BB2) = BB1 + BB2

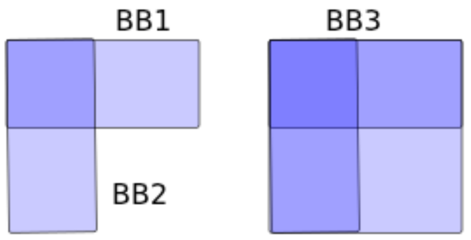


bb_intersection(BB1,BB2) = BB3

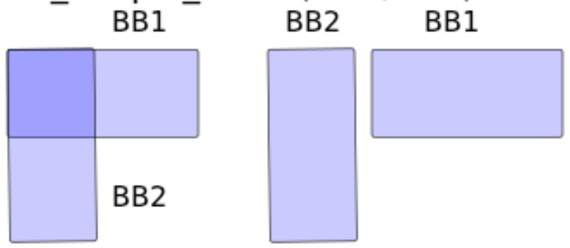


# Set Operations on Bounding Boxes

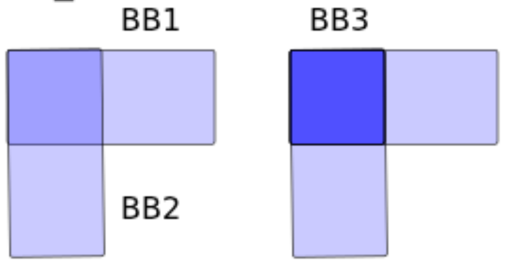
bb_convex_union(BB1,BB2) = BB3



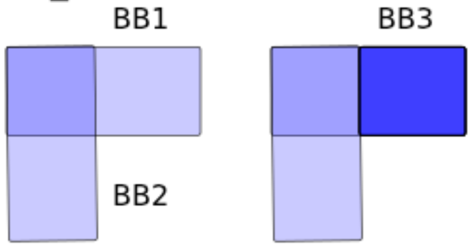
bb_simple_union(BB1,BB2) = BB1 + BB2



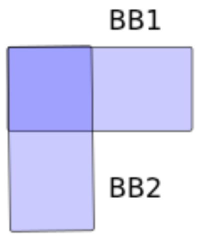
bb_intersection(BB1,BB2) = BB3



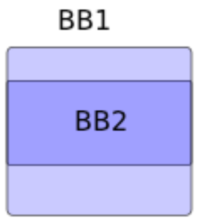
bb_subtract(BB1,BB2) = BB3



bb_is_subset (BB1,BB2)=No

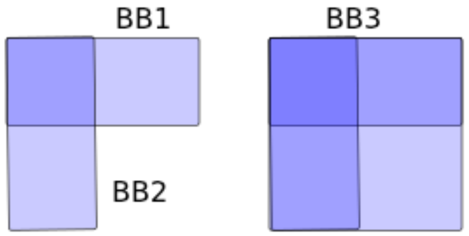


bb_is_subset (BB1,BB2)=Yes

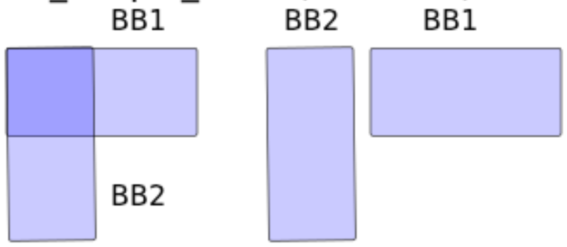


# Set Operations on Bounding Boxes

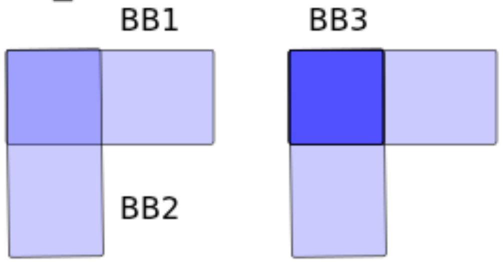
bb_convex_union(BB1,BB2) = BB3



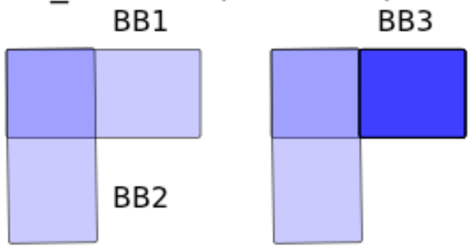
bb_simple_union(BB1,BB2) = BB1 + BB2



bb_intersection(BB1,BB2) = BB3

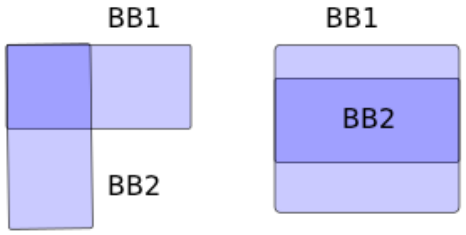


bb_subtract(BB1,BB2) = BB3



bb_is_subset (BB1,BB2)=No

bb_is_subset (BB1,BB2)=Yes



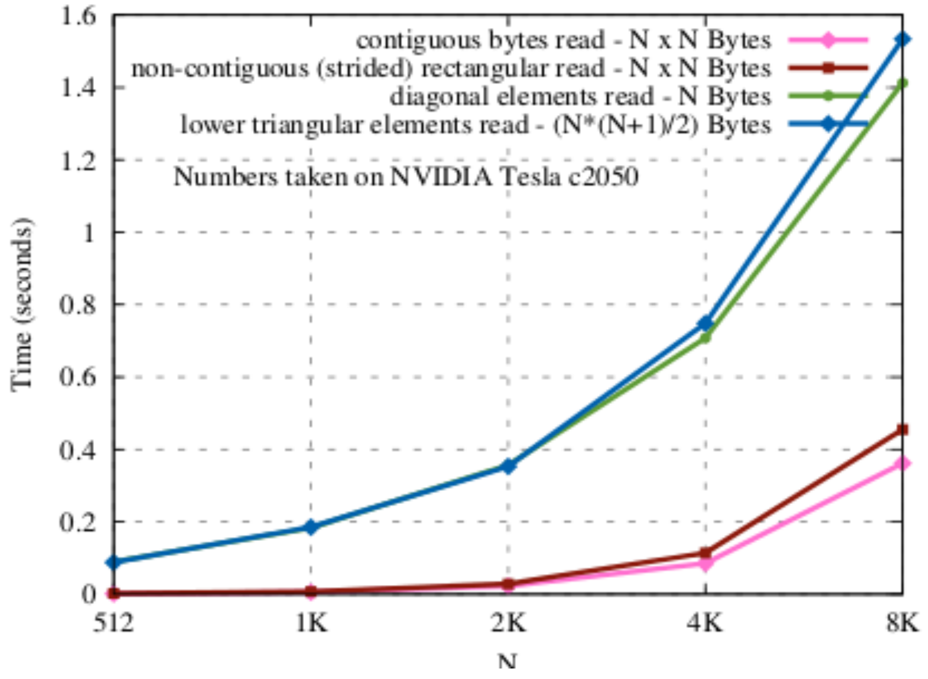
Negligible runtime overhead

# Architectural support for rectangular transfers

- Architectural support for rectangular transfers on GPU
- Support from programming models such as OpenCL and CUDA

  eg: clEnqueueReadBufferRect()

  and clEnqueueWriteBufferRect()



## The Bounding Box based memory manager (BBMM)

- Compiler-assisted runtime scheme

## The Bounding Box based memory manager (BBMM)

- Compiler-assisted runtime scheme
- Compile-time uses static analysis to identify regions of data accessed by a loop nest in terms of bounding boxes

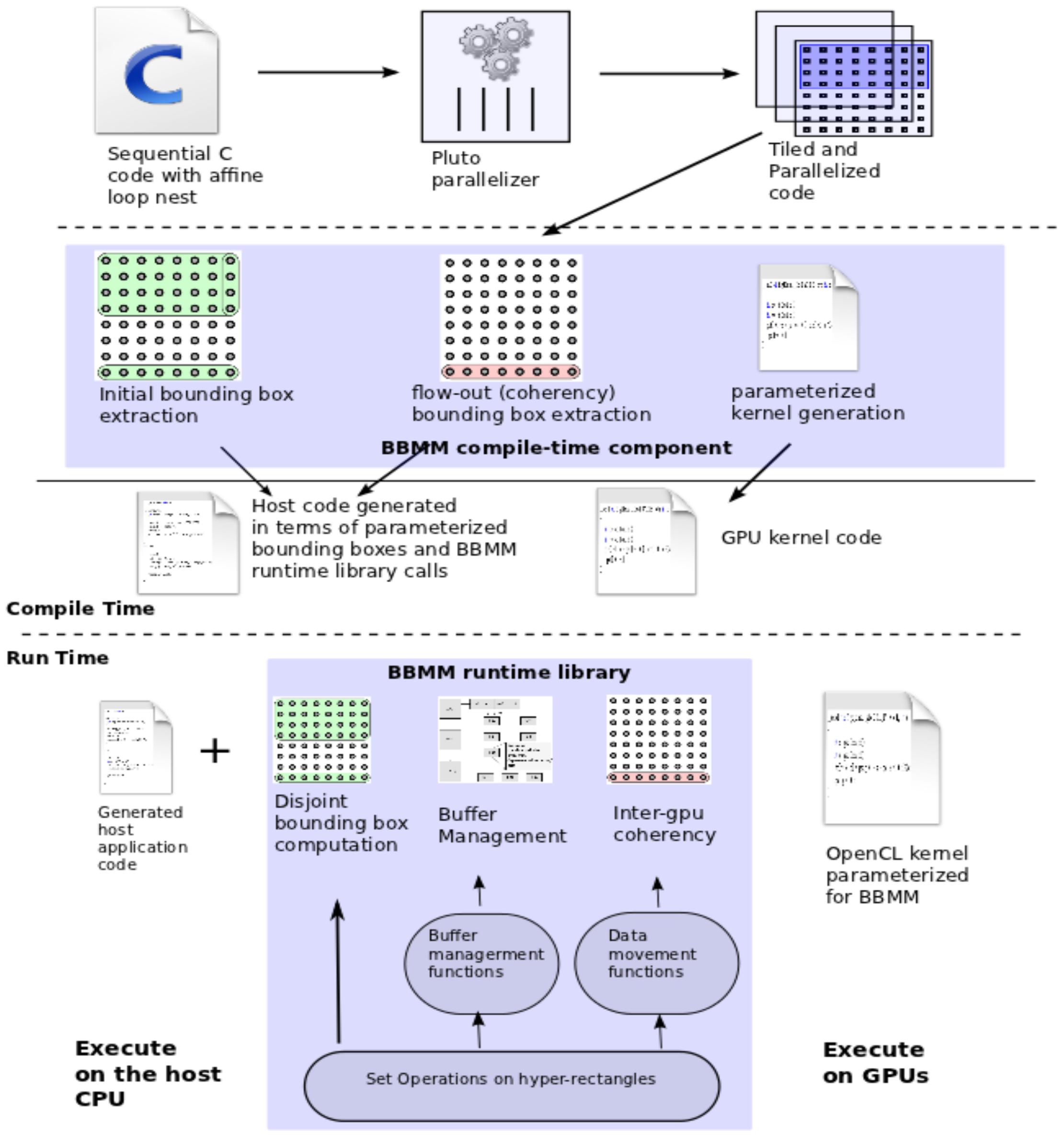
## The Bounding Box based memory manager (BBMM)

- Compiler-assisted runtime scheme
- Compile-time uses static analysis to identify regions of data accessed by a loop nest in terms of bounding boxes
- Runtime refines these initial bounding boxes into a set of disjoint bounding boxes

## The Bounding Box based memory manager (BBMM)

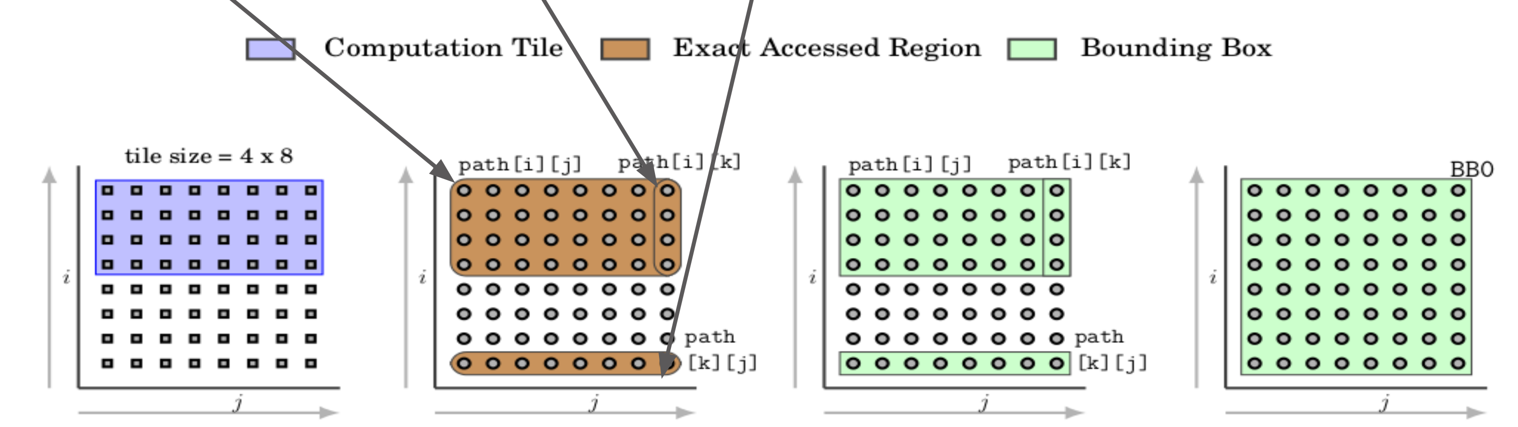
- Compiler-assisted runtime scheme
- Compile-time uses static analysis to identify regions of data accessed by a loop nest in terms of bounding boxes
- Runtime refines these initial bounding boxes into a set of disjoint bounding boxes
- All data transfers are done in terms of bounding boxes

# Overview of BBMM


Sequential C code with affine loop nest
Pluto parallelizer
Tiled and Parallelized code
Initial bounding box extraction
flow-out (coherency) bounding box extraction
parameterized kernel generation
BBMM compile-time component
Host code generated in terms of parameterized bounding boxes and BBMM runtime library calls
GPU kernel code
Compile Time
Run Time
BBMM runtime library
Generated host application code
+
Disjoint bounding box computation
Buffer Management
Inter-gpu coherency
OpenCL kernel parameterized for BBMM
Buffer management functions
Data movement functions
Set Operations on hyper-rectangles
Execute on the host CPU
Execute on GPUs

# Data allocation scheme

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]+path[k][j]:path[i][j];
```

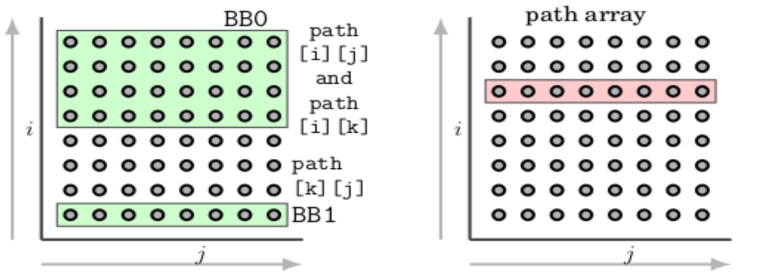


(a) Iteration space of a tile (size: 4 x 8, $k$=7)

(b) Exact array regions accessed by the tile

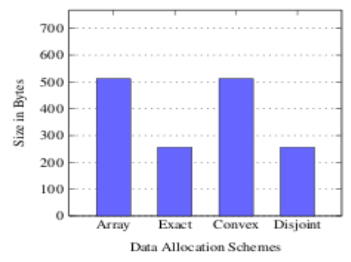
(c) Initial bounding boxes

(d) Convex bounding box

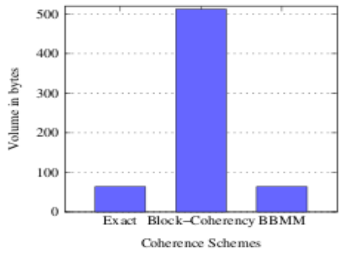


(e) Disjoint bounding boxes

(f) flow-out (coherency) bounding box (N=8,k=1)



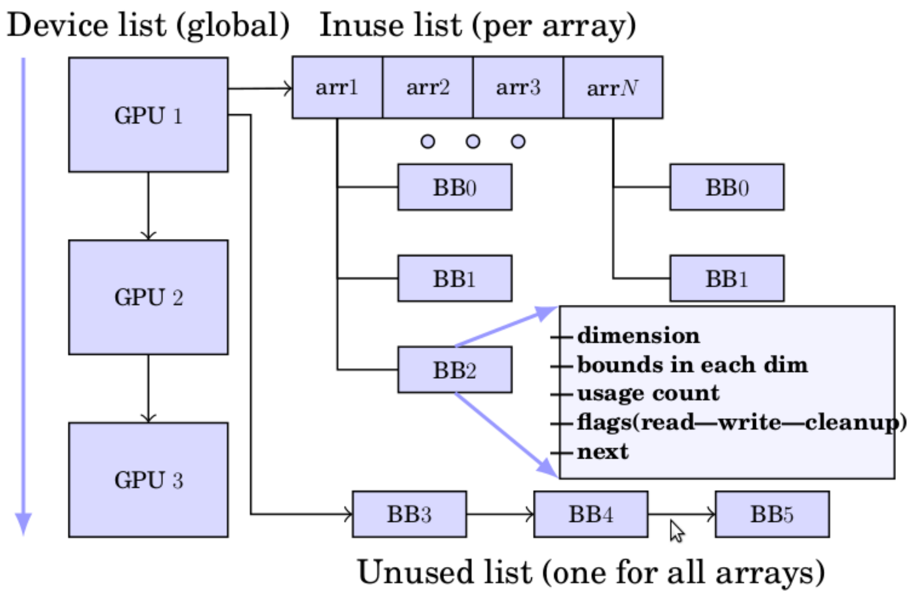
(g) Per-tile data allocation size comparison



(h) Per-iteration coherency volume comparison

## Buffer Management

- Two lists per GPU
    - inuse list
    - unused list
- Each bounding box has an associated usage count
- Flags to indicate read-only/read-write etc



## Important features of the Buffer Manager

- Inter-tile data reuse
  - Reuse data already present on the GPU
- Box-in/box-out
  - Ability to make space on the GPU when it runs out of memory

## Inter-GPU coherency

- Based on our previous work:

  *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In ACM PACT 2013.*

## Inter-GPU coherency

- Based on our previous work:

  *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In ACM PACT 2013.*

- Identify the data to be communicated from a source tile due to flow (RAW) dependences called the *Flow-out* set

## Inter-GPU coherency

- Based on our previous work:

  *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In ACM PACT 2013.*

- Identify the data to be communicated from a source tile due to flow (RAW) dependences called the *Flow-out* set
- Further refine the Flow-out set using a technique called *source-distinct-partitioning*

## Inter-GPU coherency

- Based on our previous work:

  *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In ACM PACT 2013.*

- Identify the data to be communicated from a source tile due to flow (RAW) dependences called the *Flow-out* set
- Further refine the Flow-out set using a technique called *source-distinct-partitioning*
- Eliminates both unnecessary and duplicate data transfers

## Inter-GPU coherency

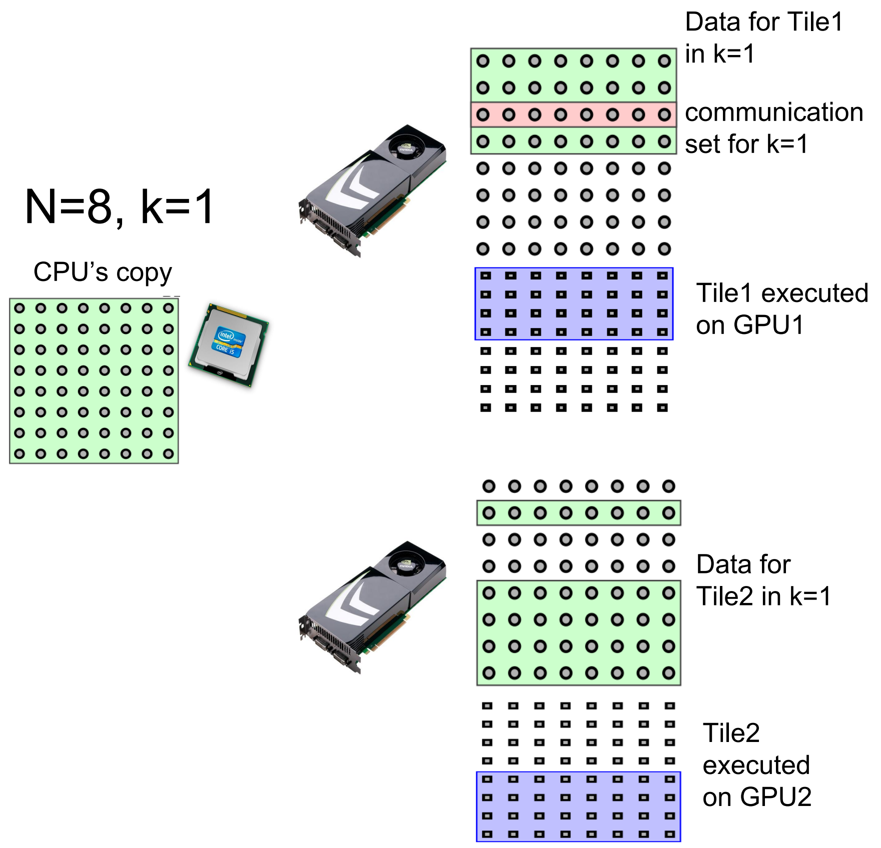
- Based on our previous work:

  *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In ACM PACT 2013.*

- Identify the data to be communicated from a source tile due to flow (RAW) dependences called the *Flow-out* set
- Further refine the Flow-out set using a technique called *source-distinct-partitioning*
- Eliminates both unnecessary and duplicate data transfers
- The scheme has been demonstrated to work well on both distributed memory and heterogeneous systems

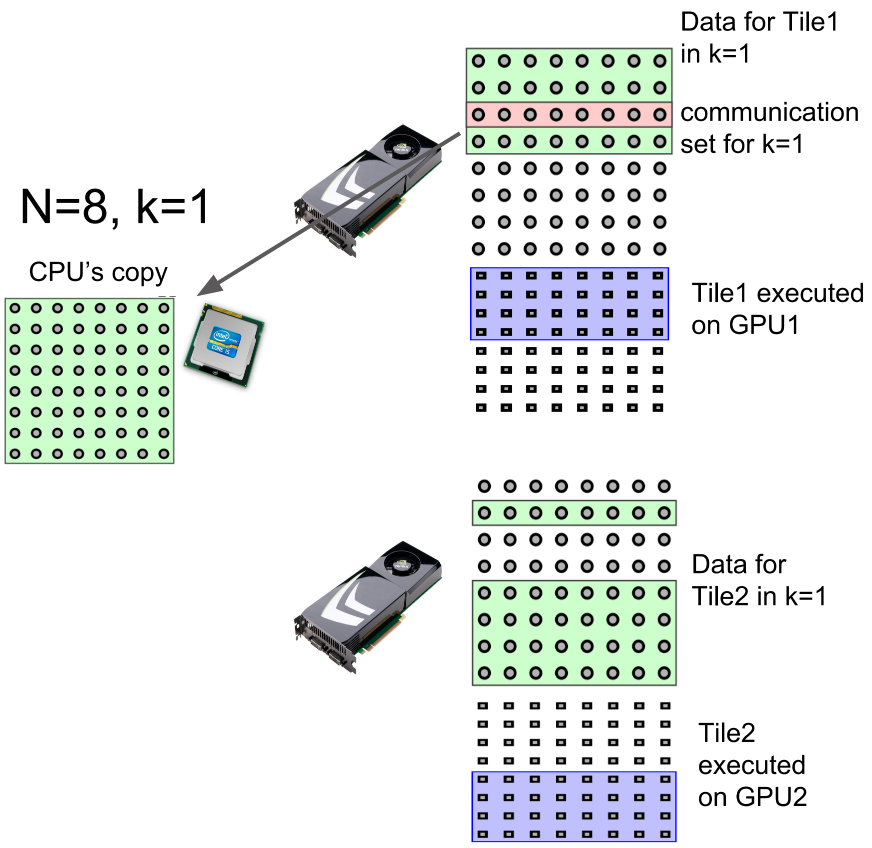
# Inter-GPU coherency (cont)

- BBMM extracts the flow-out sets as flow-out bounding boxes



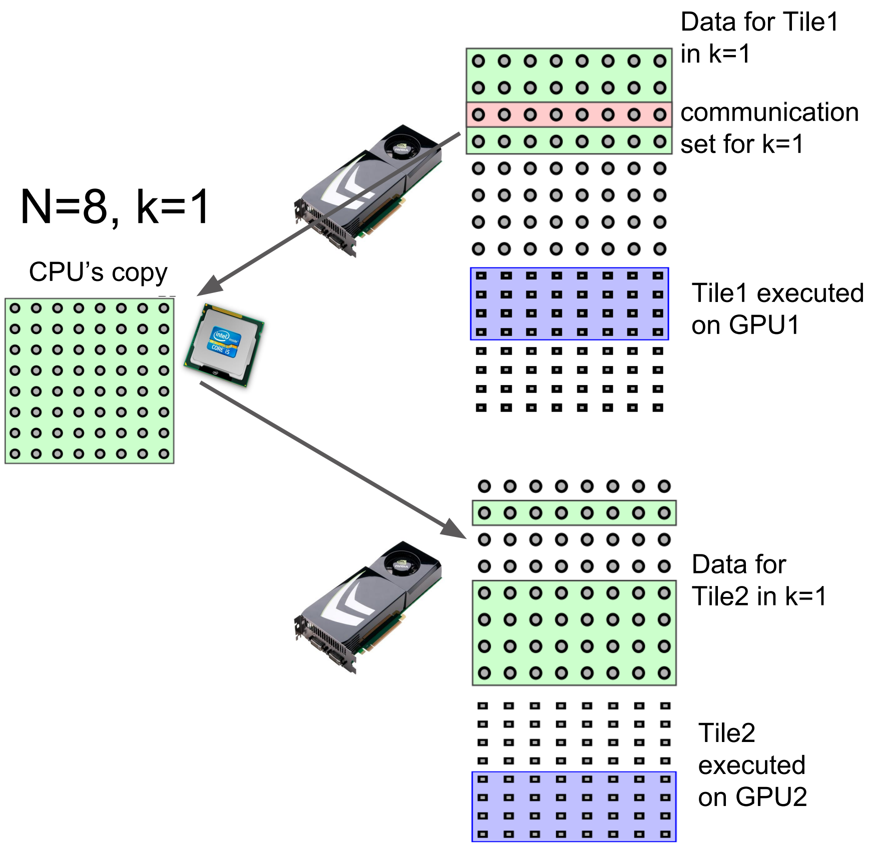
# Inter-GPU coherency (cont)

- BBMM extracts the flow-out sets as flow-out bounding boxes
- The flow-out bounding box of a tile is copied out from the source GPU onto the host CPU



# Inter-GPU coherency (cont)

- BBMM extracts the flow-out sets as flow-out bounding boxes
- The flow-out bounding box of a tile is copied out from the source GPU onto the host CPU
- If any other GPU contains the same bounding box, it is updated with a flow-in transfer
- If no GPU currently has that bounding box, the updated data is retained on the CPU



## Implementation

- The compile-time component integrated into polyhedral source-to-source transformer - Pluto

## Implementation

- The compile-time component integrated into polyhedral source-to-source transformer - Pluto
- The input to the compile-time is the sequential C code containing a set of affine loop nests

## Implementation

- The compile-time component integrated into polyhedral source-to-source transformer - Pluto
- The input to the compile-time is the sequential C code containing a set of affine loop nests
- Pluto creates a tiled and parallelized version of the input code

## Implementation

- The compile-time component integrated into polyhedral source-to-source transformer - Pluto
- The input to the compile-time is the sequential C code containing a set of affine loop nests
- Pluto creates a tiled and parallelized version of the input code
- BBMM's compile-time component takes this tiled and parallelized code as input and generates the following:
    - A set of initial and flow-out bounding boxes
    - The code similar to the host code structure shown in Algorithm 4.

## Implementation

- The compile-time component integrated into polyhedral source-to-source transformer - Pluto
- The input to the compile-time is the sequential C code containing a set of affine loop nests
- Pluto creates a tiled and parallelized version of the input code
- BBMM's compile-time component takes this tiled and parallelized code as input and generates the following:
    - A set of initial and flow-out bounding boxes
    - The code similar to the host code structure shown earlier
- The runtime component is implemented as stand-alone C library.

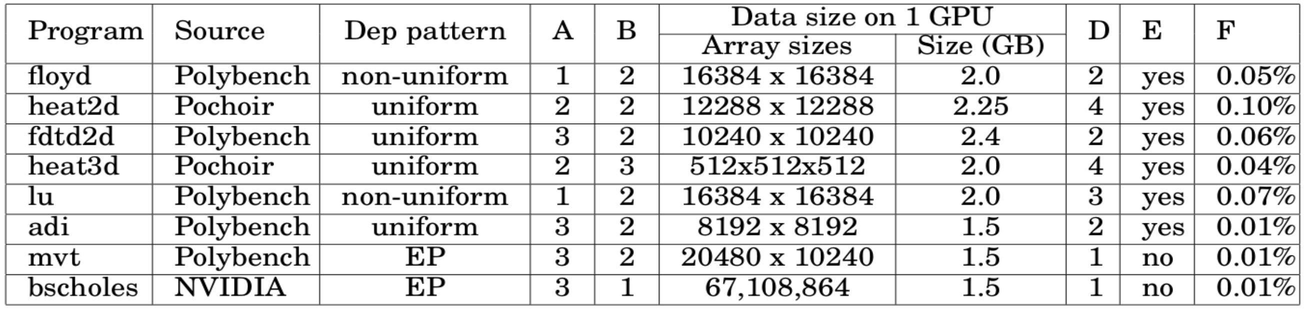
## Evaluation and Results

## Evaluation and Results

- Setup
  - A multi-GPU machine consisting of 3 NVIDIA Tesla c2050 (fermi) GPUs and 1 NVIDIA Tesla K20 (Kepler) with 2.5 GB of memory each
  - A 12-core CPU system as the host

# Evaluation and Results

- Setup
  - A multi-GPU machine consisting of 3 NVIDIA Tesla c2050 (fermi) GPUs and 1 NVIDIA Tesla K20 (Kepler) with 2.5 GB of memory each
  - A 12-core CPU system as the host
- Benchmarks



| Program | Source | Dep pattern | A | B | Data size on 1 GPU | | D | E | F |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | Array sizes | Size (GB) | | | |
| floyd | Polybench | non-uniform | 1 | 2 | 16384 x 16384 | 2.0 | 2 | yes | 0.05% |
| heat2d | Pochoir | uniform | 2 | 2 | 12288 x 12288 | 2.25 | 4 | yes | 0.10% |
| fdtd2d | Polybench | uniform | 3 | 2 | 10240 x 10240 | 2.4 | 2 | yes | 0.06% |
| heat3d | Pochoir | uniform | 2 | 3 | 512x512x512 | 2.0 | 4 | yes | 0.04% |
| lu | Polybench | non-uniform | 1 | 2 | 16384 x 16384 | 2.0 | 3 | yes | 0.07% |
| adi | Polybench | uniform | 3 | 2 | 8192 x 8192 | 1.5 | 2 | yes | 0.01% |
| mvt | Polybench | EP | 3 | 2 | 20480 x 10240 | 1.5 | 1 | no | 0.01% |
| bscholes | NVIDIA | EP | 3 | 1 | 67,108,864 | 1.5 | 1 | no | 0.01% |

A: number of arrays. B: maximum dimensionality of arrays C: bounding box type chosen by our algorithm D: maximum number of bounding boxes for any array E: subsumed bounding boxes present? F: BBMM runtime overhead as a percentage of overall execution time

## Evaluation Parameters

## Evaluation Parameters

- Overhead of the runtime library

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes
- Performance with data scaling

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes
- Performance with data scaling
- Comparison with manually written code

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes
- Performance with data scaling
- Comparison with manually written code
- Performance with box-in/box-out

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes
- Performance with data scaling
- Comparison with manually written code
- Performance with box-in/box-out
- Benefits of inter-tile data reuse

## Evaluation Parameters

- Overhead of the runtime library
- Comparison of data allocation sizes
- Performance with data scaling
- Comparison with manually written code
- Performance with box-in/box-out
- Benefits of inter-tile data reuse
- Performance with access function split

## Overhead of runtime library

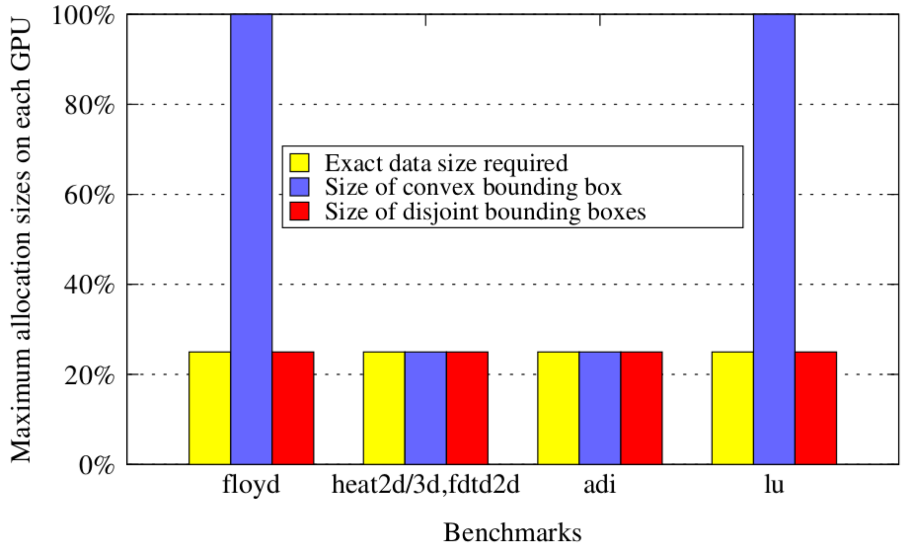
*total_execution_time = memory_mgmt_time + compute_time + flowout_time + flowin_time + writeout_time*

*overhead_percentage = (memory_mgmt_time / total_execution_time) * 100*

- For all programs, the runtime overhead was less than 0.1% of the total execution time of the program (hence insignificant)

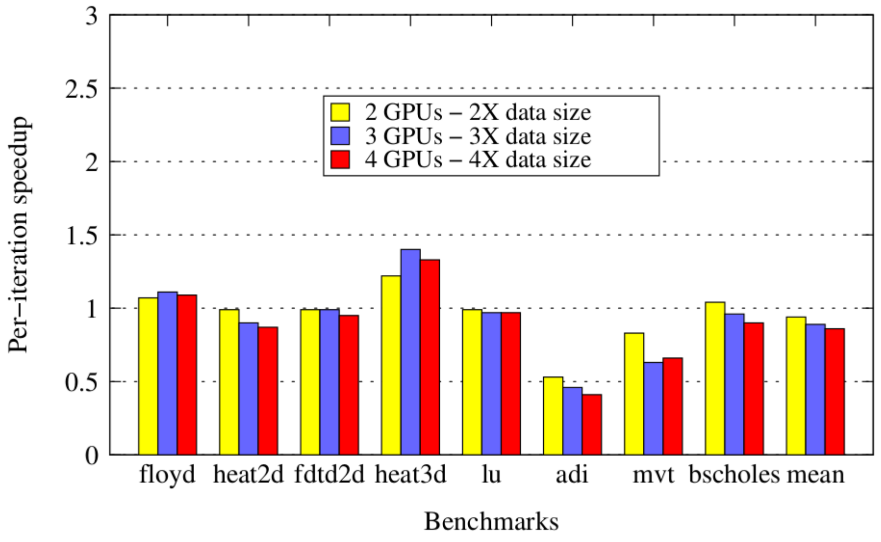
## Comparison of data allocation sizes

- Up to 75% reduction on a 4-GPU machine compared to convex bounding box scheme
- Equal to the exact data sizes required (manually computed) for all cases



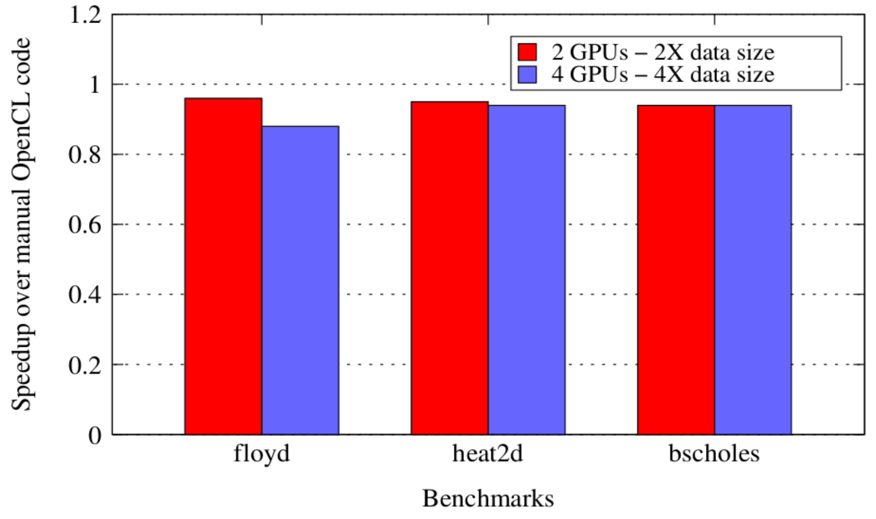
## Performance with data scaling

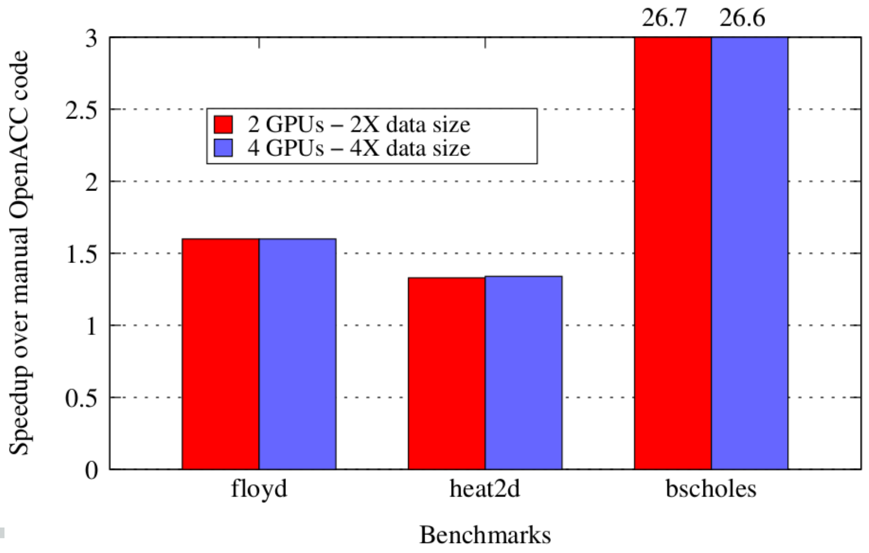
- Data scaling similar to weak scaling but with emphasis on data size (memory utilization) rather than on problem size (computation)
- Hence we consider the per-iteration speedup
- The per-iteration time includes all overhead: data allocation time, compute time, flow-out time, flow-in time and write-out time
- BBMM affects all the above except compute time
- Mean speedup of 0.94 indicating near-ideal speedup



# Comparison with manually written code

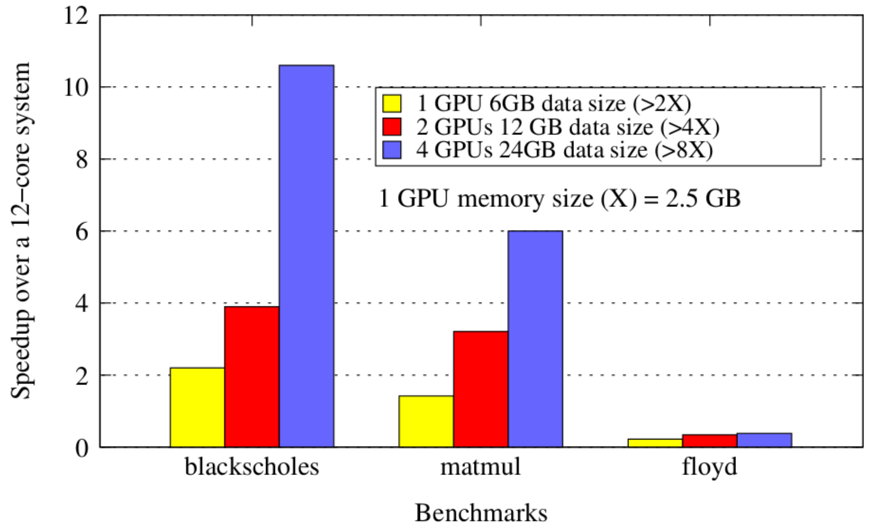
- Manual code has following optimizations:
  - Optimized to have theoretically minimum data allocation sizes and coherency volume
  - Reuse exploitation was theoretical maximum
- BBMM at least 88% as efficient as manual OpenCL code
- Outperforms the manual OpenACC code

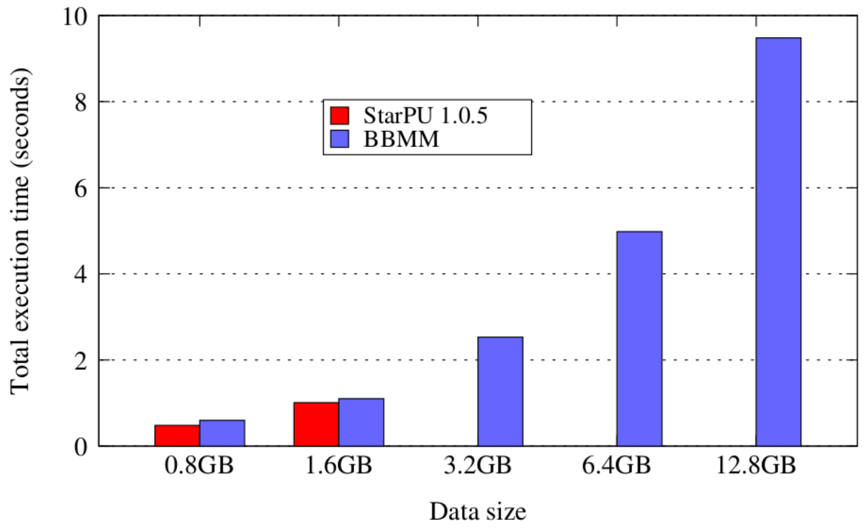




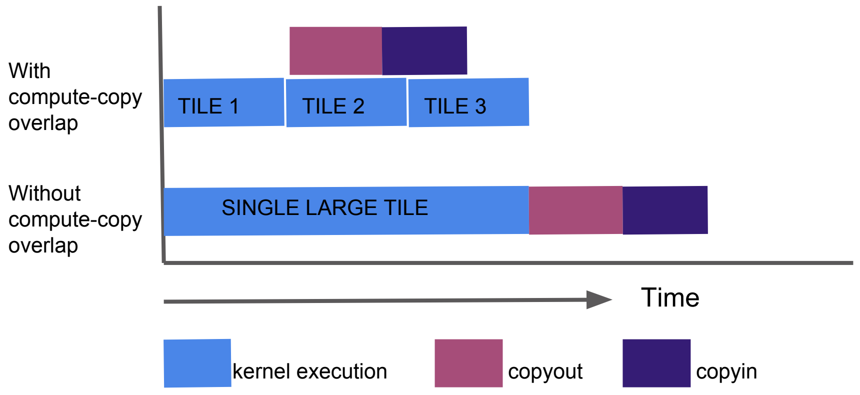
# Benefit of box-in/box-out

- Significant performance improvements with tiles that have sufficient compute-to-copy ratio
- Without it, significant performance degradation
- With right tiling strategy, the feature can allow applications to work with data sizes significantly larger than available GPU memory



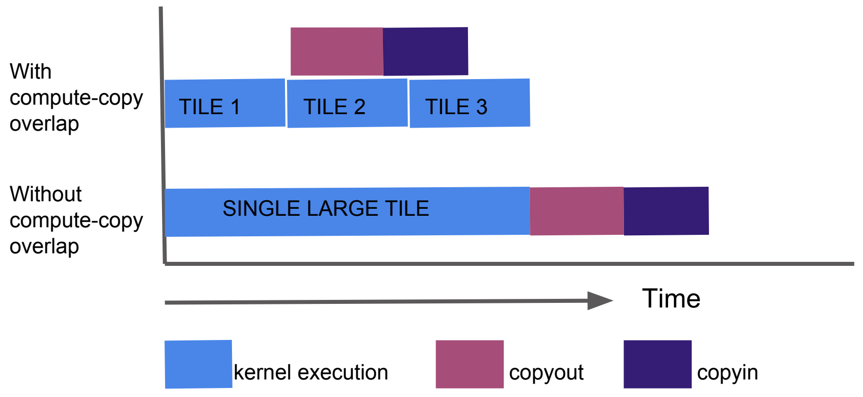


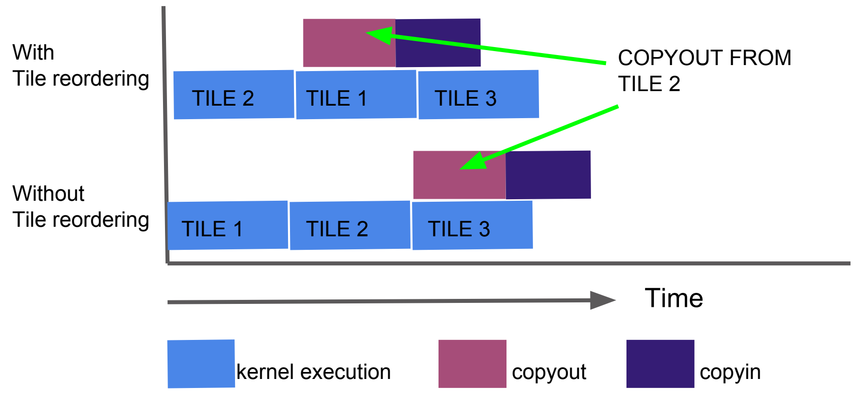
## Compute-Copy Overlap



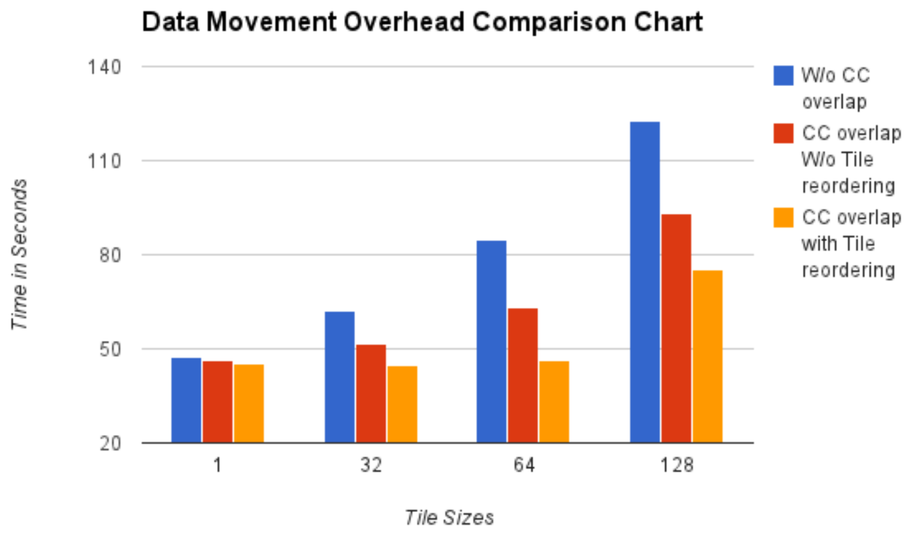
- Hide the data movement overhead within computation time
- Split the computation allocated to a GPU into multiple tiles
- Register a callback to be called at the completion of each tile
- In the callback perform the CopyOut() and CopyIn()
- CopyIn() does not conflict because we work on a distributed parallel loop

# Maximizing Compute-Copy Overlap

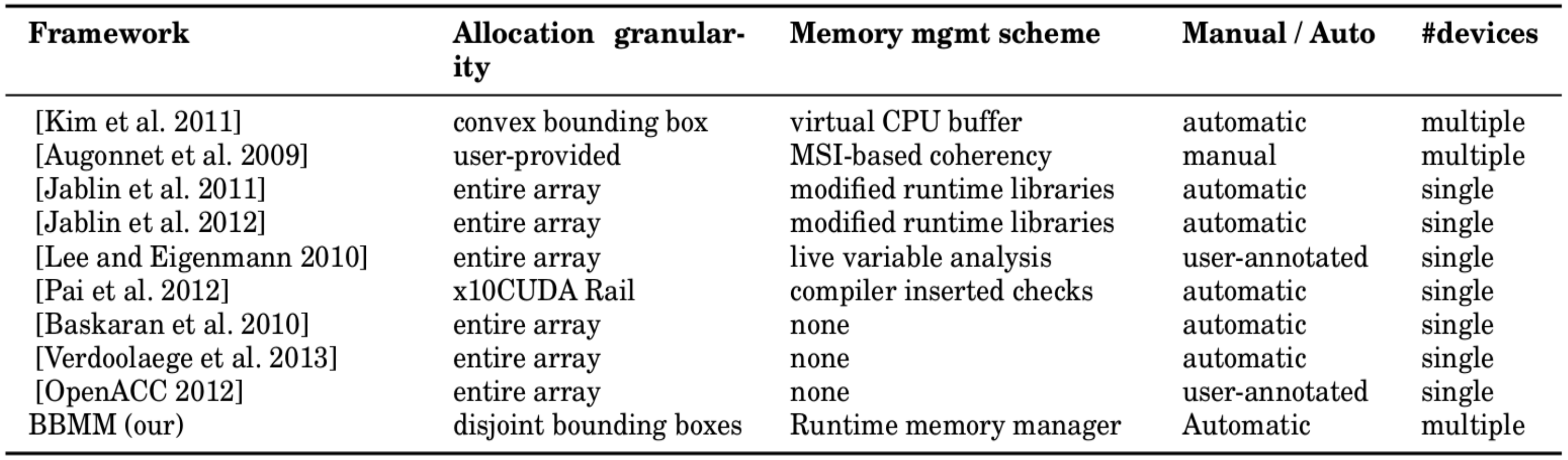




- Sort the tiles based on size of the CopyOut data
- Schedule them in the sorted order (largest copyout size first)



## Related Work



| Framework | Allocation granular-ity | Memory mgmt scheme | Manual / Auto | #devices |
|---|---|---|---|---|
| [Kim et al. 2011] | convex bounding box | virtual CPU buffer | automatic | multiple |
| [Augonnet et al. 2009] | user-provided | MSI-based coherency | manual | multiple |
| [Jablin et al. 2011] | entire array | modified runtime libraries | automatic | single |
| [Jablin et al. 2012] | entire array | modified runtime libraries | automatic | single |
| [Lee and Eigenmann 2010] | entire array | live variable analysis | user-annotated | single |
| [Pai et al. 2012] | x10CUDA Rail | compiler inserted checks | automatic | single |
| [Baskaran et al. 2010] | entire array | none | automatic | single |
| [Verdoolaege et al. 2013] | entire array | none | automatic | single |
| [OpenACC 2012] | entire array | none | user-annotated | single |
| BBMM (our) | disjoint bounding boxes | Runtime memory manager | Automatic | multiple |

## Conclusion

- We presented a fully automatic data allocation and memory management framework for affine loop nests on multi-GPU machines
- Data allocation, buffer management, inter-GPU coherency were all done at the granularity of bounding boxes

## Conclusion

- We presented a fully automatic data allocation and memory management framework for affine loop nests on multi-GPU machines
- Data allocation, buffer management, inter-GPU coherency were all done at the granularity of bounding boxes
- On a 4-GPU machine our scheme was able to:
  - Achieve allocation size reductions of 75% compared existing schemes
  - Comparison to manual OpenCL and OpenACC code showed:
    - Our code yielded a performance of at least 88% of manual OpenCL code
    - Outperformed OpenACC code in all the cases
  - Achieve excellent data scaling

## Conclusion

- We presented a fully automatic data allocation and memory management framework for affine loop nests on multi-GPU machines
- Data allocation, buffer management, inter-GPU coherency were all done at the granularity of bounding boxes
- On a 4-GPU machine our scheme was able to:
    - Achieve allocation size reductions of 75% compared existing schemes
    - Comparison to manual OpenCL and OpenACC code showed:
        - Our code yielded a performance of at least 88% of manual OpenCL code
        - Outperformed OpenACC code in all the cases
    - Achieve excellent data scaling
- All the above achieved with an insignificant runtime overhead of 0.1%

## Conclusion

- We presented a fully automatic data allocation and memory management framework for affine loop nests on multi-GPU machines
- Data allocation, buffer management, inter-GPU coherency were all done at the granularity of bounding boxes
- On a 4-GPU machine our scheme was able to:
  - Achieve allocation size reductions of 75% compared existing schemes
  - Comparison to manual OpenCL and OpenACC code showed:
    - Our code yielded a performance of at least 88% of manual OpenCL code
    - Outperformed OpenACC code in all the cases
  - Achieve excellent data scaling
- All the above achieved with an insignificant runtime overhead of 0.1%
- Our work is suited to any compiler/runtime system targeting GPUs
- Can bridge the data allocation gap that exists in programming these systems

## Publications based on this work

1. *Automatic Data Allocation and Buffer Management for Multi-GPU Machines*

   *Thejas Ramashekar, Uday Bondhugula, In the ACM Transactions on Architecture and Code Optimization, Vol. 10, No. 4, Article 60, Publication date: December 2013 . Selected for presentation at HiPEAC '14, Jan 2014, Vienna, Austria.*

2. *Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory*

   *Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula, Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2013.*

## References

Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *CC 2010*.

Cédric Bastoul. 2005. Clan: The Chunky Loop Analyzer. (2005).

Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *ACM/IEEE Supercomputing (SC '13)*. ACM, Denver, Colorado, USA.

Uday Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN PLDI*.

Daniel Chavarría-Miranda and John Mellor-Crummey. 2005. Effective communication coalescing for data-parallel applications. In *ACM SIGPLAN PPoPP*.

M. Classen and M. Griebl. 2006. Automatic code generation for distributed memory architectures in the polytope model. In *IEEE IPDPS*.

CUDA 2011. NVIDIA CUDA. (2011). http://developer.nvidia.com/object/cuda.html

Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed Memory. In *PACT 2013*.

Armin Größlinger. 2009. Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes. In *Compiler Construction*. 236–250.

ISL 2012. Integer Set Library. (2012). Sven Verdoolaege, An Integer Set Library for Program Analysis.

Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically Managed Data for CPU-GPU Architectures. In *CGO*.

Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. In *ACM PLDI*.

Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *ACM SIGPLAN PPoPP*.

Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A Hybrid Approach of OpenMP for Clusters. In *PPoPP*. http://engineering.purdue.edu/paramnt/publications/ppopp12.pdf

Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC 2010*.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *ACM SIGPLAN PPoPP*.

NVIDIA GPU Computing SDK 2010. NVIDIA GPU Computing SDK. (2010). https://developer.nvidia.com/gpu-computing-sdk

OpenACC 2012. OpenACC Application Programming Interface. (2012). http://www.openacc-standard.org/

OpenCL 2011. OpenCL. (2011). http://www.khronos.org/opencl/

Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2012. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *ACM PACT*.

# References

Vikram S. Adve and John M. Mellor-Crummey. 1998. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *ACM SIGPLAN PLDI*. 186–198.

Saman P. Amarasinghe and Monica S. Lam. 1993. Communication optimization and code generation for distributed memory machines. In *PLDI*. 126–138.

C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Concurrency and Computation: Practice and Experience*.

M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *ACM SIGPLAN PPoPP*.

## Backup Slides

# Data Allocation Scheme Algorithms

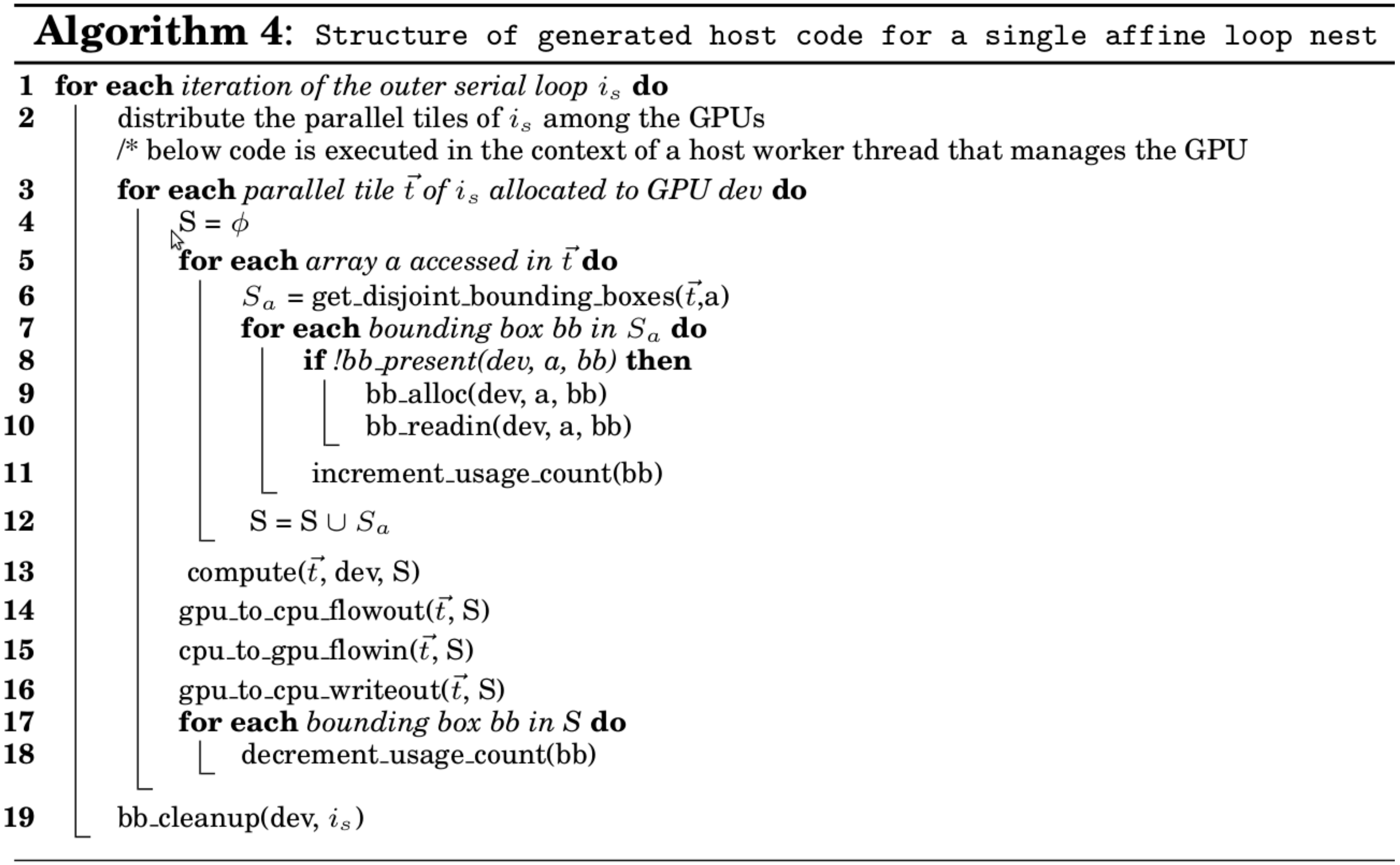
**Algorithm 1:** `extract_initial_bounding_boxes()`

```
Input: Computation tile $\vec{t}$, Array a
$S_a^{init} = \phi$
for each read or write access function $f_a^i$ do
    $dp_a^i$ = get_data_polyhedron($f_a^i$)
    $bb_a^i$ = get_bounding_box($dp_a^i$)
    add $bb_a^i$ to $S_a^{init}$
Output: $S_a^{init}$, the set of initial bounding boxes
```

**Algorithm 2:** `get_disjoint_bounding_boxes()`

```
Input: $S_a^{init}$ - Set of initial bounding boxes for tile $\vec{t}$ and array a
$S_a^{disjoint} = \phi$
for each bounding box $bb_a^{init}$ in $S_a^{init}$ do
    $bb_a^{rem} = bb_a^{init}$
    for each bounding box $bb_a^{disj}$ in $S_a^{disjoint}$ do
        $bb_a^{intersect}$ = bb_intersection($bb_a^{rem}$, $bb_a^{disj}$)
        $bb_a^{rem}$ = bb_subtract($bb_a^{rem}$, $bb_a^{intersect}$)
    add $bb_a^{rem}$ to $S_a^{disjoint}$
Output: $S_a^{disjoint}$, the set of disjoint bounding boxes for array a
```

# Structure of the generated host code



**Algorithm 4:** `Structure of generated host code for a single affine loop nest`

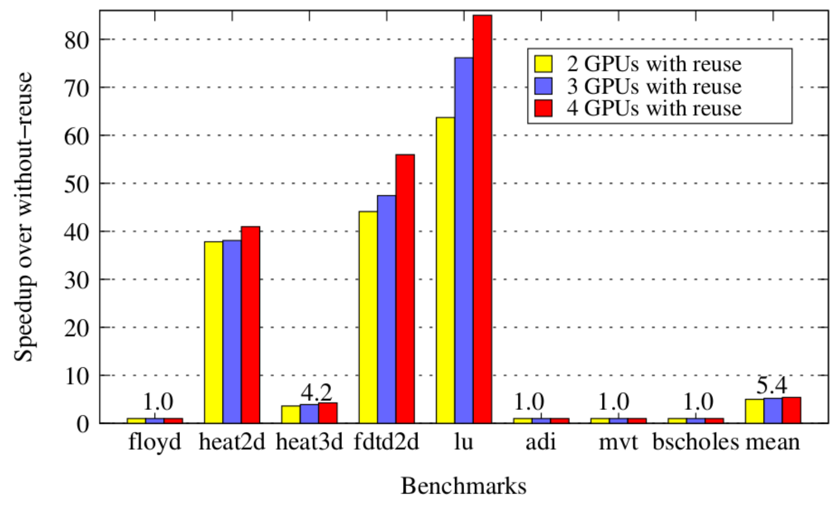
```
for each iteration of the outer serial loop $i_s$ do
    distribute the parallel tiles of $i_s$ among the GPUs
    /* below code is executed in the context of a host worker thread that manages the GPU
    for each parallel tile $\vec{t}$ of $i_s$ allocated to GPU dev do
        S = $\phi$
        for each array a accessed in $\vec{t}$ do
            $S_a$ = get_disjoint_bounding_boxes($\vec{t}$,a)
            for each bounding box bb in $S_a$ do
                if !bb_present(dev, a, bb) then
                    bb_alloc(dev, a, bb)
                    bb_readin(dev, a, bb)
                increment_usage_count(bb)
            S = S $\cup$ $S_a$
        compute($\vec{t}$, dev, S)
        gpu_to_cpu_flowout($\vec{t}$, S)
        cpu_to_gpu_flowin($\vec{t}$, S)
        gpu_to_cpu_writeout($\vec{t}$, S)
        for each bounding box bb in S do
            decrement_usage_count(bb)
    bb_cleanup(dev, $i_s$)
```

## Structure of the generated kernel code

```
void ComputeKernel0(int split0, DATA_TYPE * buf0, int buf0_lb0, int buf0_ub0, int buf0_lb1, int buf0_ub1,
      int split1, DATA_TYPE * buf1, int buf1_lb0, int buf1_ub0, int buf1_lb1, int buf1_ub1, ....)
{
    DATA_TYPE * var_wacc_0 = KERNEL0_var_WACC(split0, buf0, buf0_lb0, buf0_ub0, buf0_lb1, buf0_ub1, idx0,
      idx1);
    DATA_TYPE var_racc_0  = KERNEL0_var_RACC(split1, buf1, buf1_lb0, buf1_ub0, buf1_lb1, buf1_ub1, idx0,
      idx1);
    ...
    // do the computation using values obtained above.
    *var_wacc_0 = var_racc_0 + ...
}
```

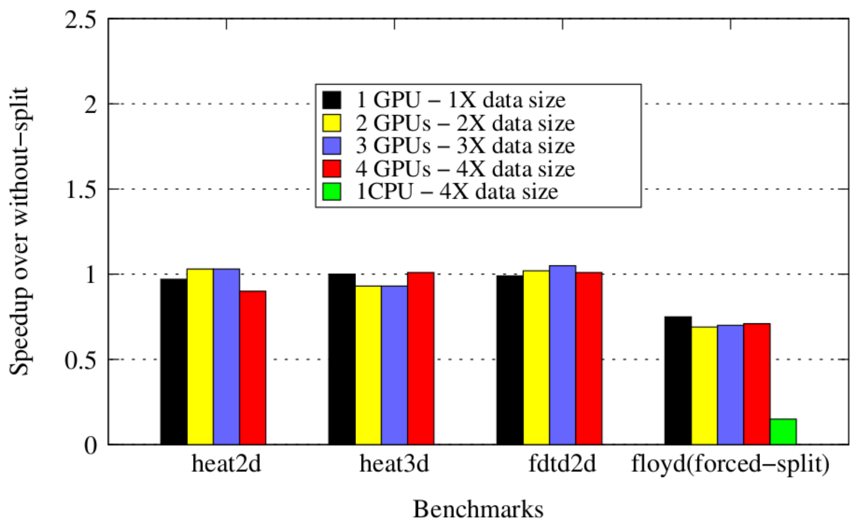
## Performance with inter-tile reuse

- compared to performance of the same code without reuse
- mean speedup of 5.4x with maximum speedup of upto 85x



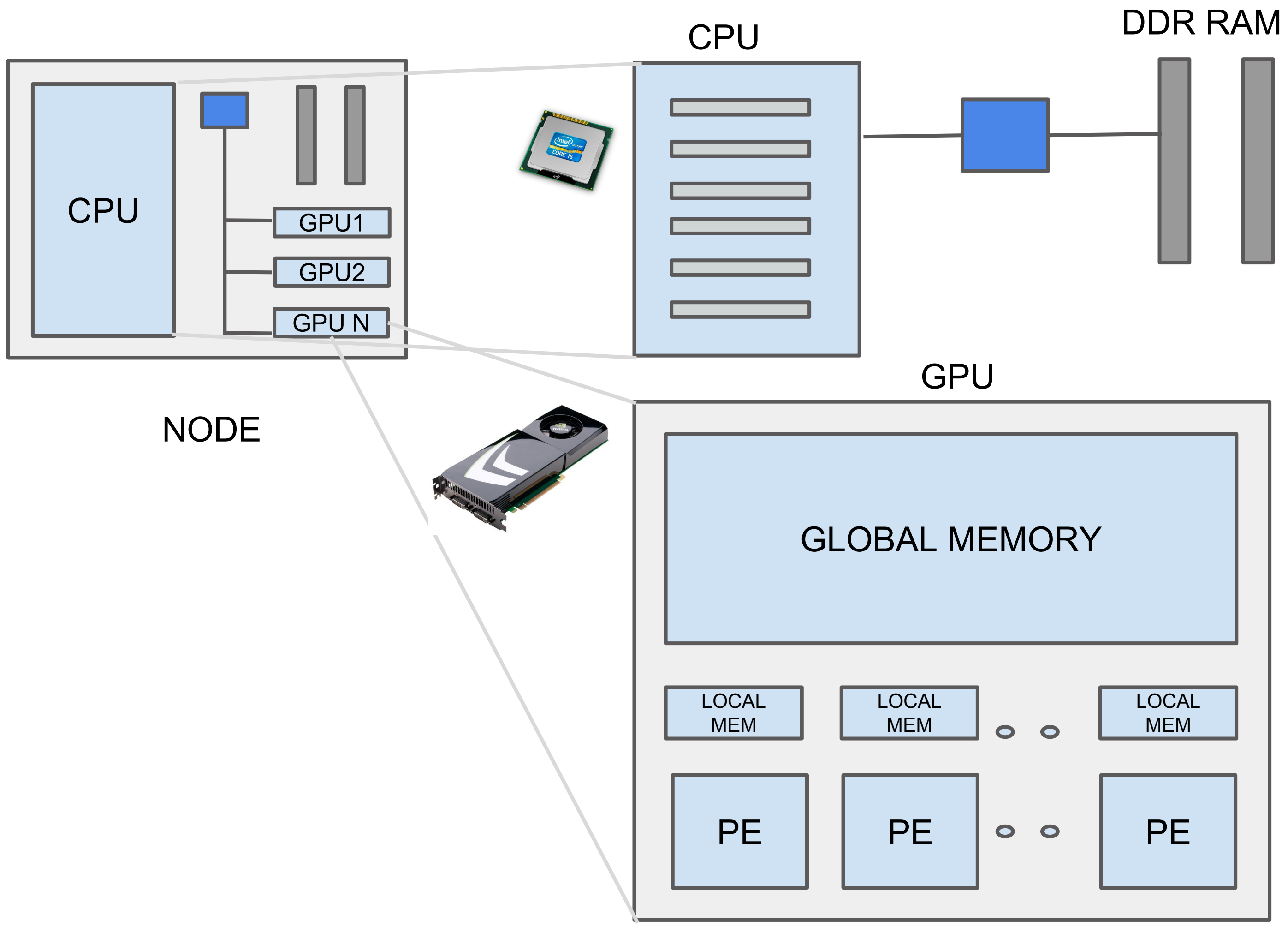
# Performance with access function split

- compared to performance of code without splits
- stencils did not undergo performance degradation
- floyd in the worst case, suffered 40% performance loss. But still much better compared to CPU execution times



## Table of contents

# Distributed memory paradigm


CPU
GPU1
GPU2
GPU N
NODE
CPU
DDR RAM
GPU
GLOBAL MEMORY
LOCAL MEM
LOCAL MEM
LOCAL MEM
PE
PE
PE