Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters

Jonathan Lifflander, G. Carl Evans, Anshu Arya, Laxmikant Kale

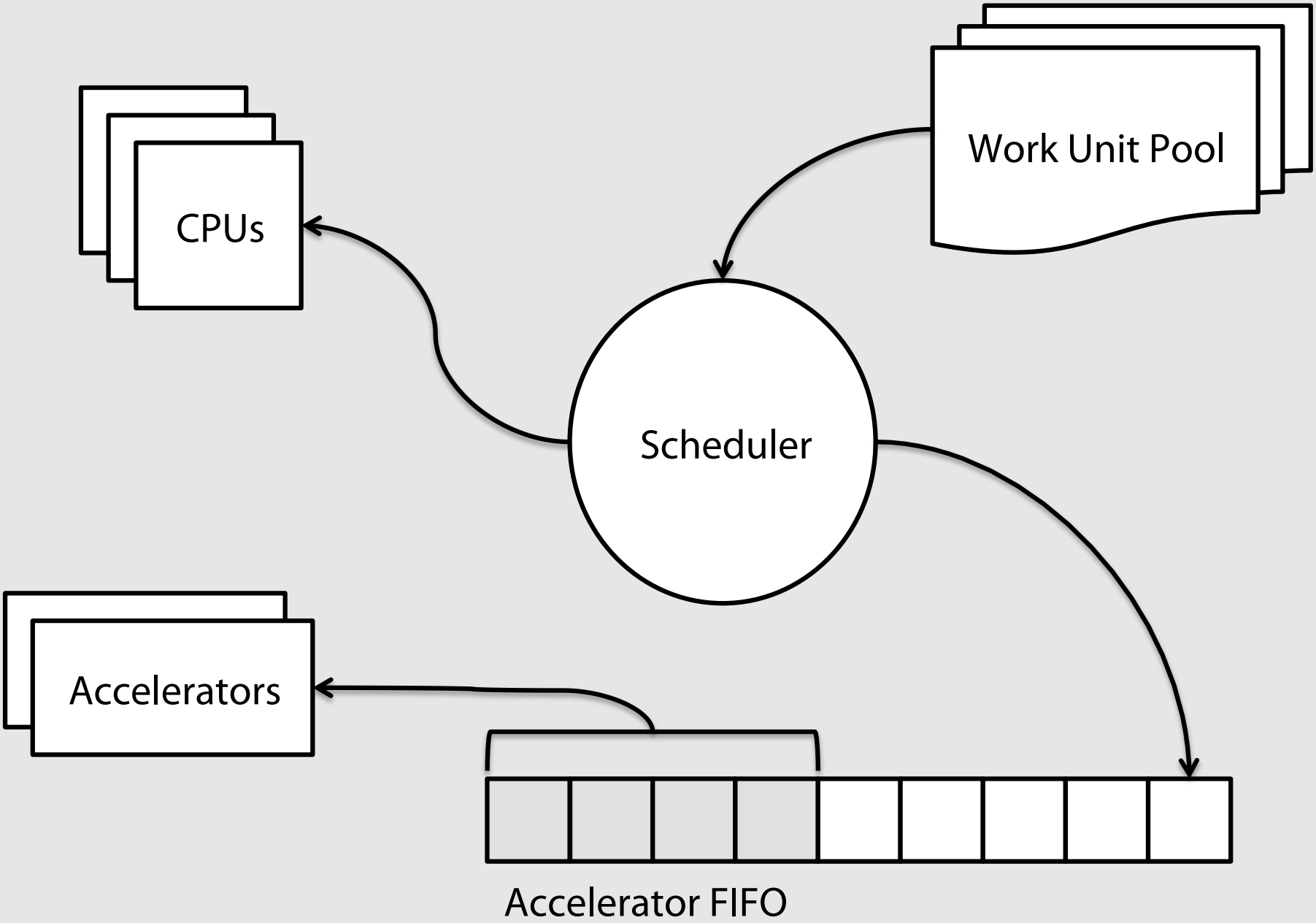
University of Illinois Urbana-Champaign

May 25, 2012

- Work is *overdecomposed* into medium-sized grains
  - Fine-grain task parallelism
  - Sized well for the CPU
    - Overlap of communication and computation
  - GPUs rely on massive data-parallelism
    - Fine grains decrease performance
    - Each kernel instantiation has substantial overhead
- To reduce overhead
  - Combine fine-grain work units for the GPU
  - Delay may be insignificant if the work is low priority

Terminology

- *Agglomeration*—composition of distinct work units
- *Static agglomeration*—fixed number of work units are agglomerated
- *Dynamic agglomeration*—number of work units agglomerated varies at runtime

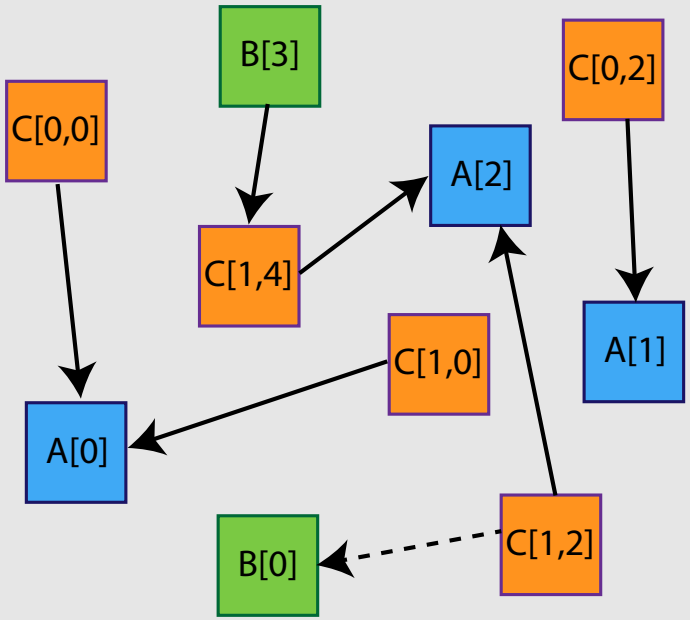

CPUs
Work Unit Pool
Scheduler
Accelerators
Accelerator FIFO

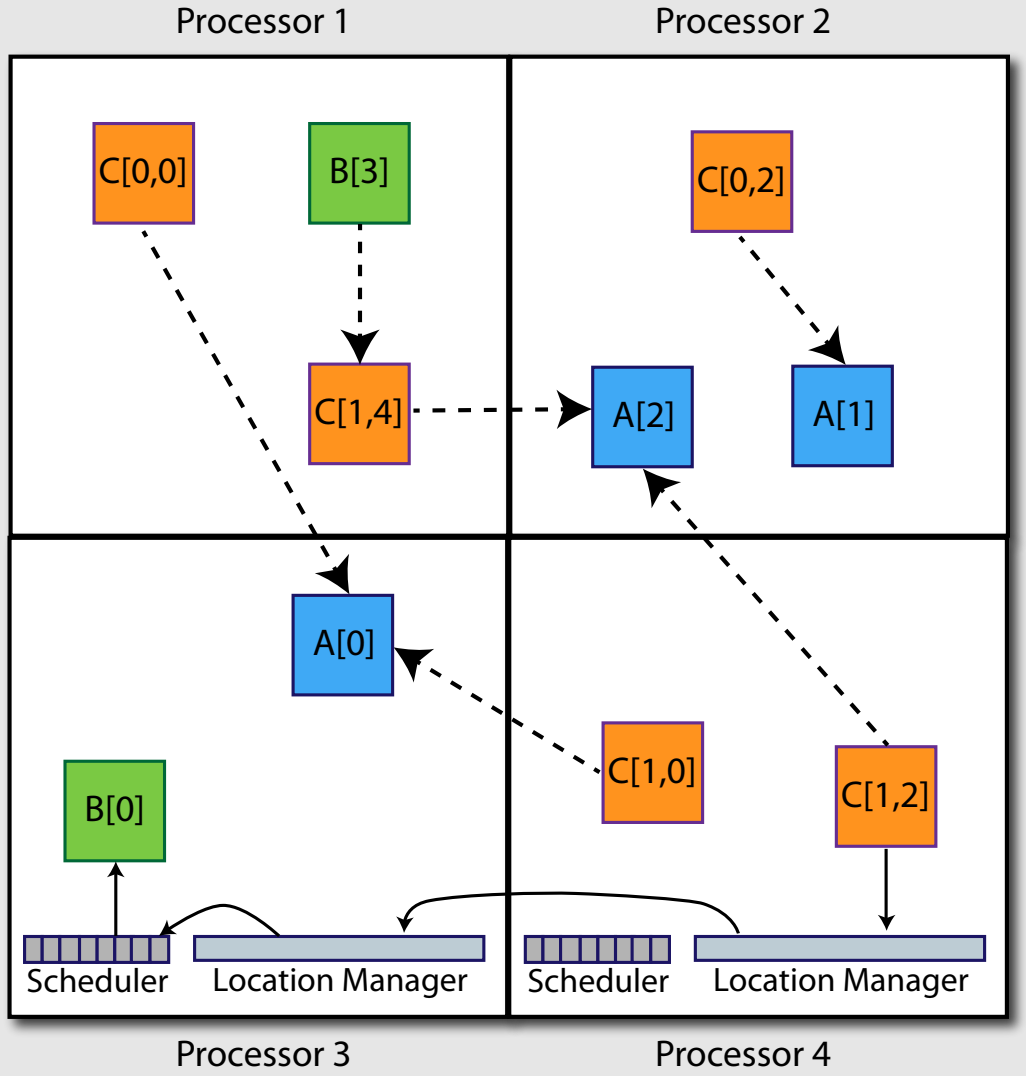
Implementation

- Charm++
  - Work is decomposed into objects
    - With affinity to data
    - Represents multiple tasks
    - Each task is a method of an object
    - Remote invocation occurs when a message arrives (the data is the parameters)
  - Each object lives on a processor
  - Each processor has many objects on it
  - Sets of objects that perform the same type of work are organized into arrays

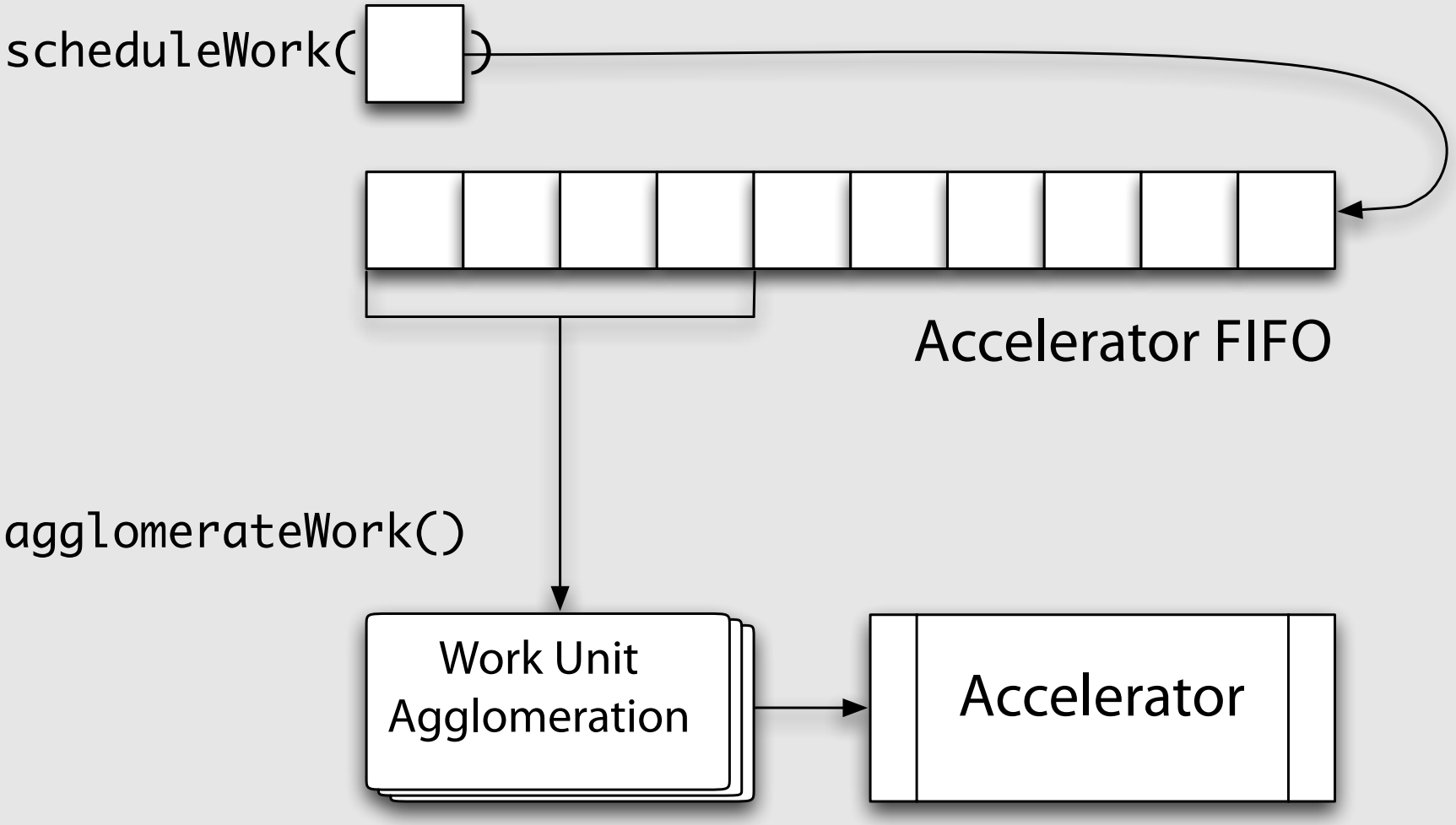
Charm++ Scheduling

- Each processor has a scheduler
  - Arrival messages are put in a queue
  - They are prioritized based on priority set by the sender
  - Execution is in that order based on the current queue state


B[3]
C[0,0]
C[0,2]
A[2]
C[1,4]
C[1,0]
A[1]
A[0]
B[0]
C[1,2]

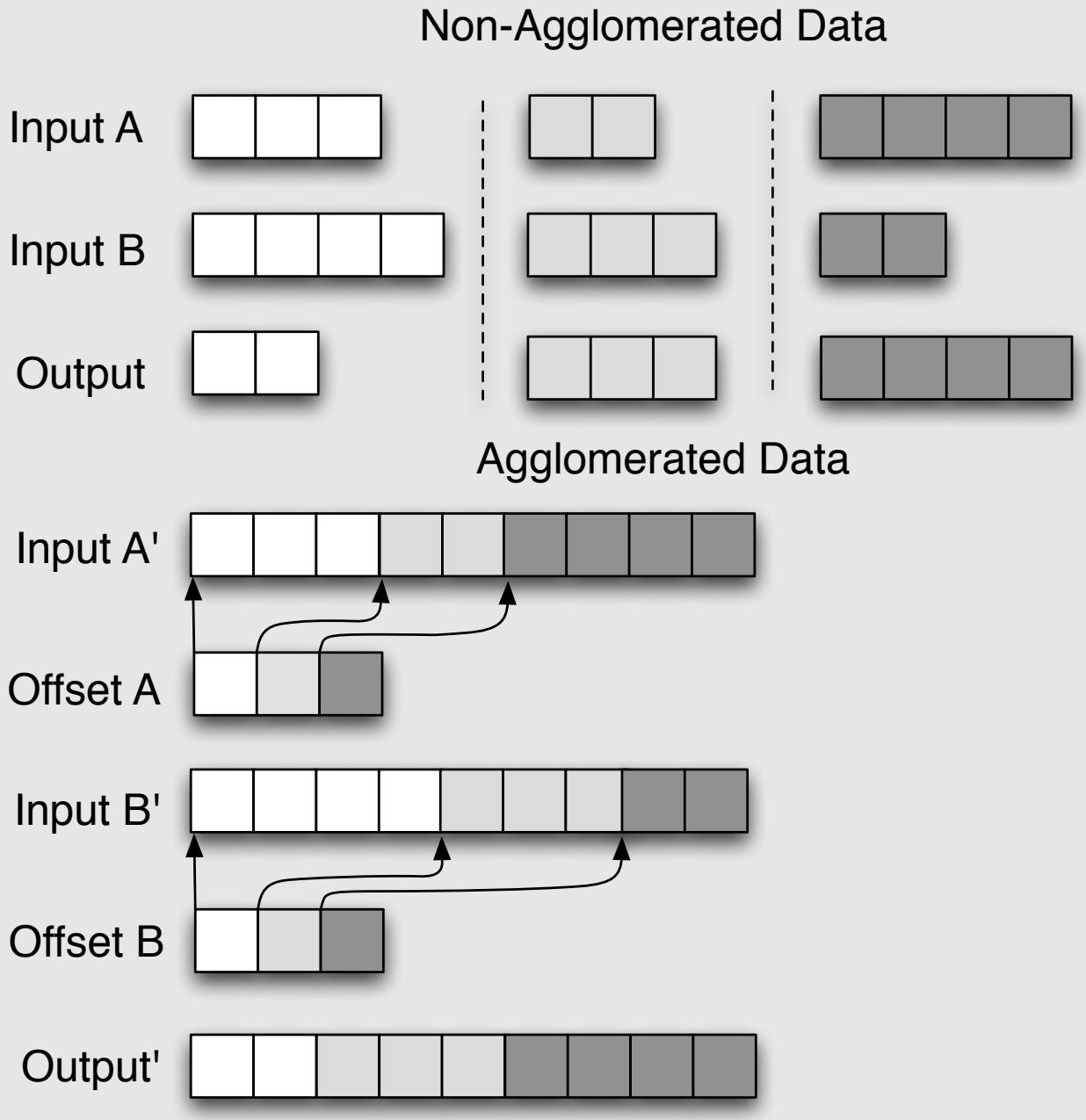

Processor 1
Processor 2
C[0,0]
B[3]
C[0,2]
C[1,4]
A[2]
A[1]
A[0]
C[1,0]
C[1,2]
B[0]
Scheduler
Location Manager
Scheduler
Location Manager
Processor 3
Processor 4

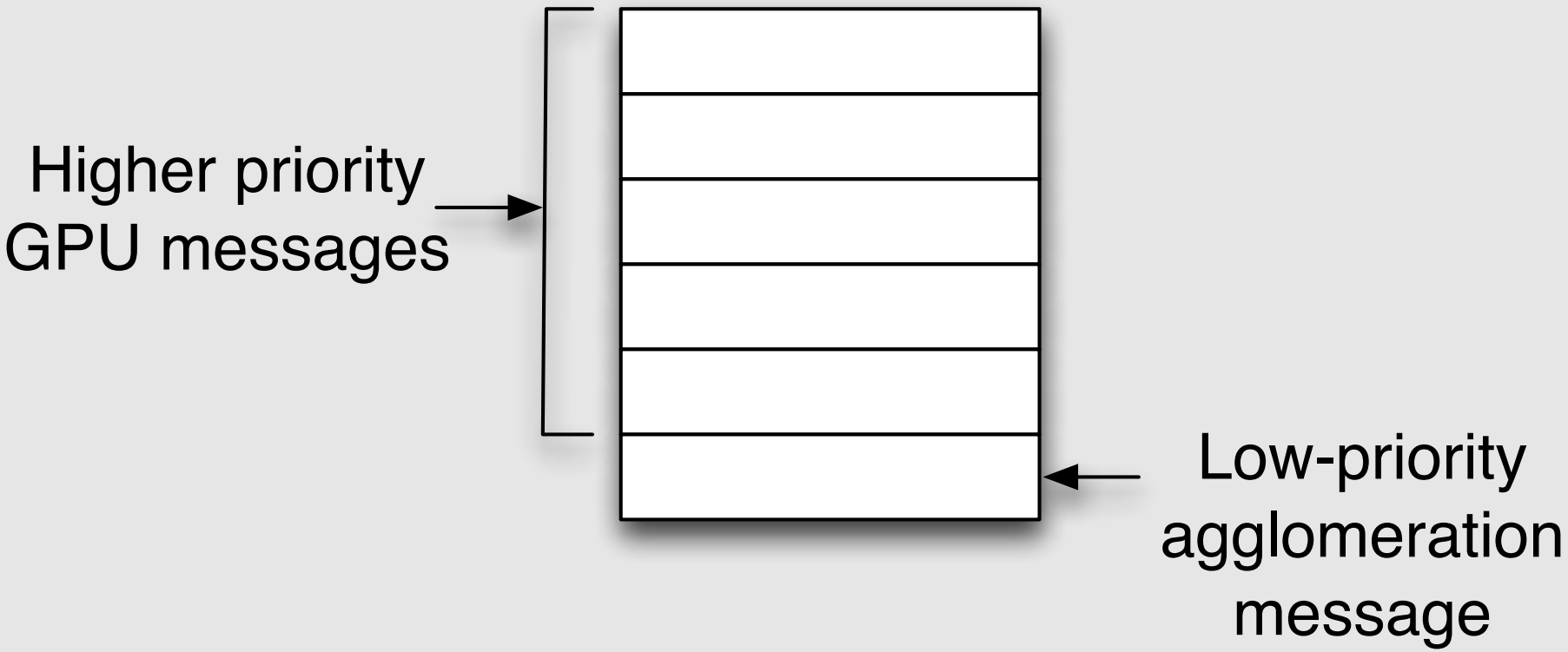
Agglomeration API



Programmer/Runtime Division

- Programmer
  - Writes GPU kernel for agglomeration
  - Creates an *offset array*
    - Each task's input might be a different size
    - Store the offset of each task's beginning and ending index in the contiguous data arrays
- System
  - Decide what work to execute and when


Non-Agglomerated Data
Input A
Input B
Output
Agglomerated Data
Input A'
Offset A
Input B'
Offset B
Output'


Higher priority
GPU messages
Low-priority
agglomeration
message

Dynamic Agglomeration

- Uses the following heuristic
  - If the “accelerator FIFO” reaches a size limit, work is agglomerated
    - Typically set based on memory limitations
  - Else enqueue a low priority message that causes agglomeration
    - When higher-priority work is being generated, it goes into the FIFO
    - When it lets up, work is agglomerated
    - Since low priority work is assumed, not agglomerating aggressively should not reduce performance

Experimental Setup

- NCSA AC Cluster
  - Two dual-core 2.4 GHz AMD Opterons
    - 8 GB of memory
  - NVIDIA Tesla S1070 with four GPUs
    - Each with 4 GB of memory
    - CUDA

Application: Molecular2D

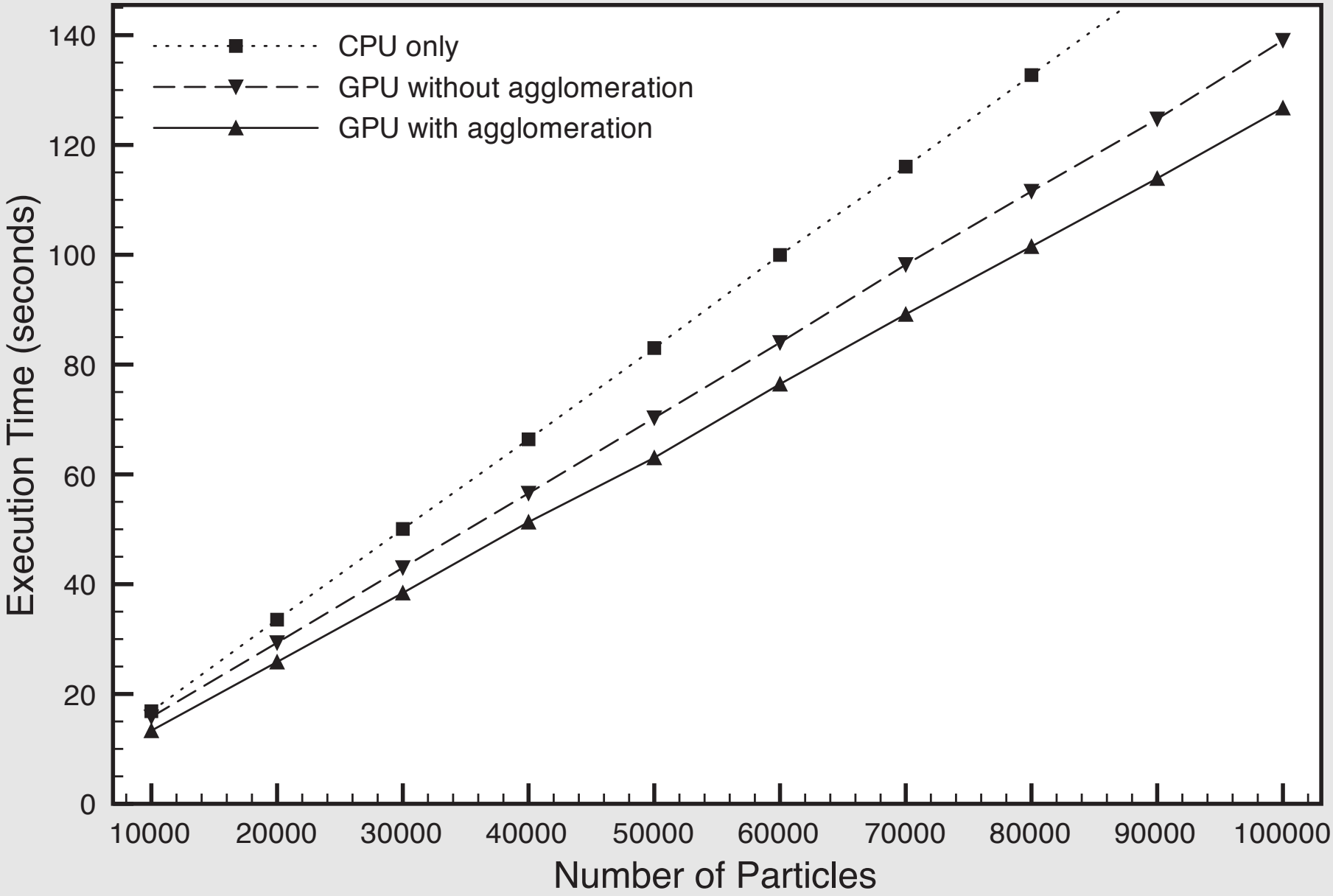
Molecular2D

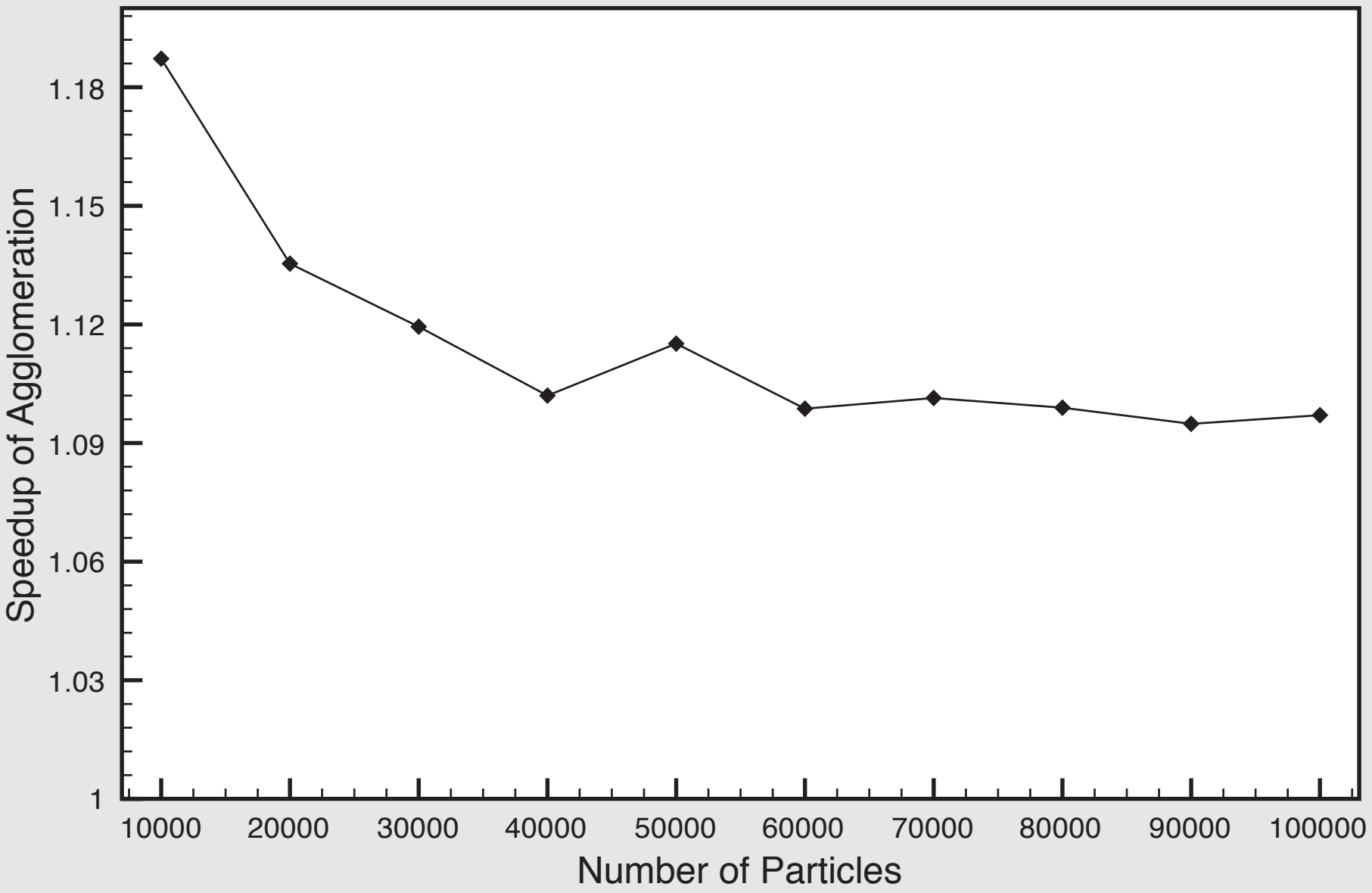
- Work is decomposed into:
  - *Cells: 2D array of objects*
    - Spatially decomposed
    - Each holds a set of particles
    - They interact with the neighboring cells
    - The cell holds the current particle position and updates these based on calculated forces
  - *Interactions: 4D array of objects*
    - Each interacts two particle sets
    - Bulk of the work
- Using the GPU
  - Cells on CPU
  - Interactions on GPU
  - When an interaction receives the two particles sets, it calls `scheduleWork`

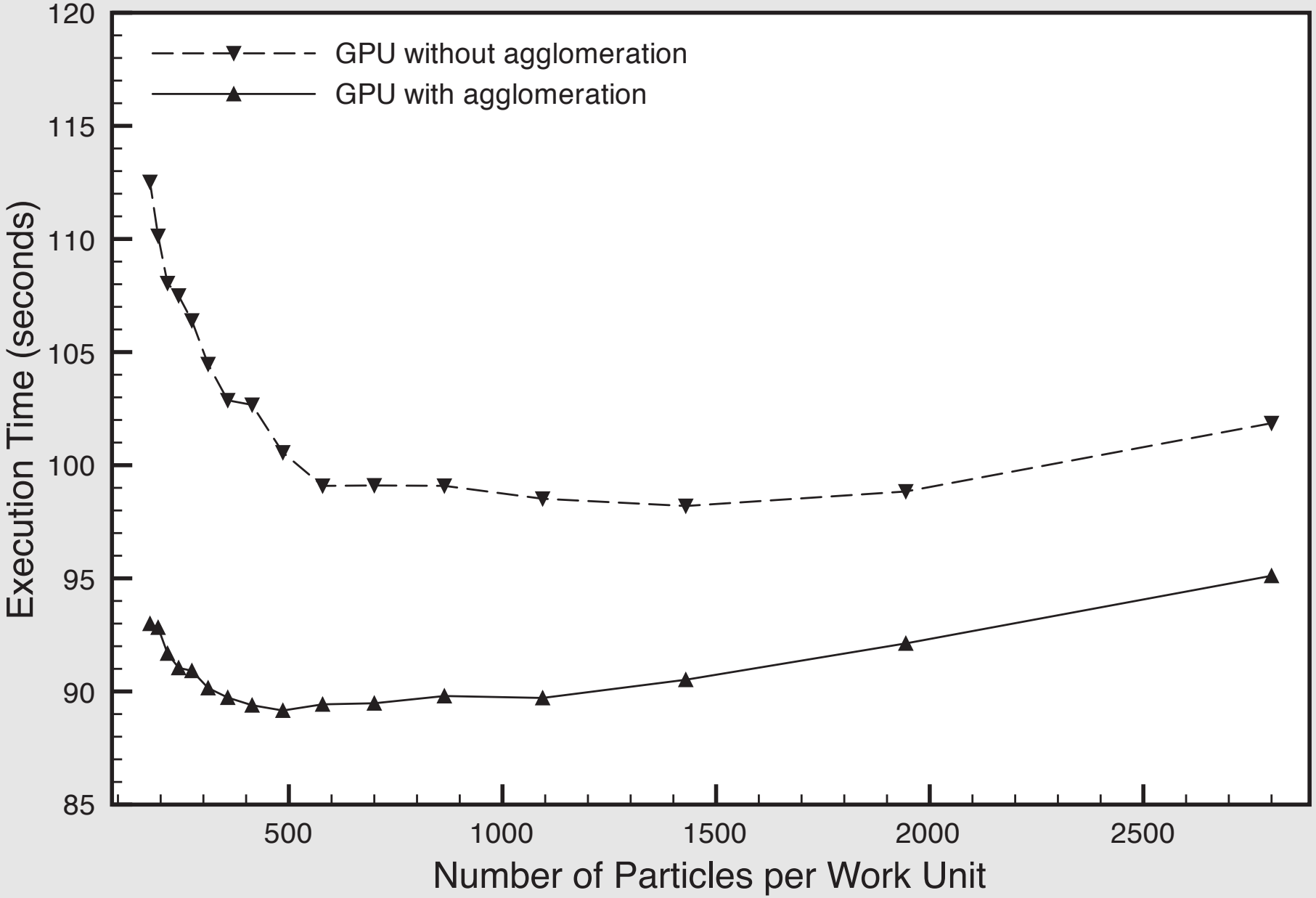
Molecular 2D Interaction Kernel

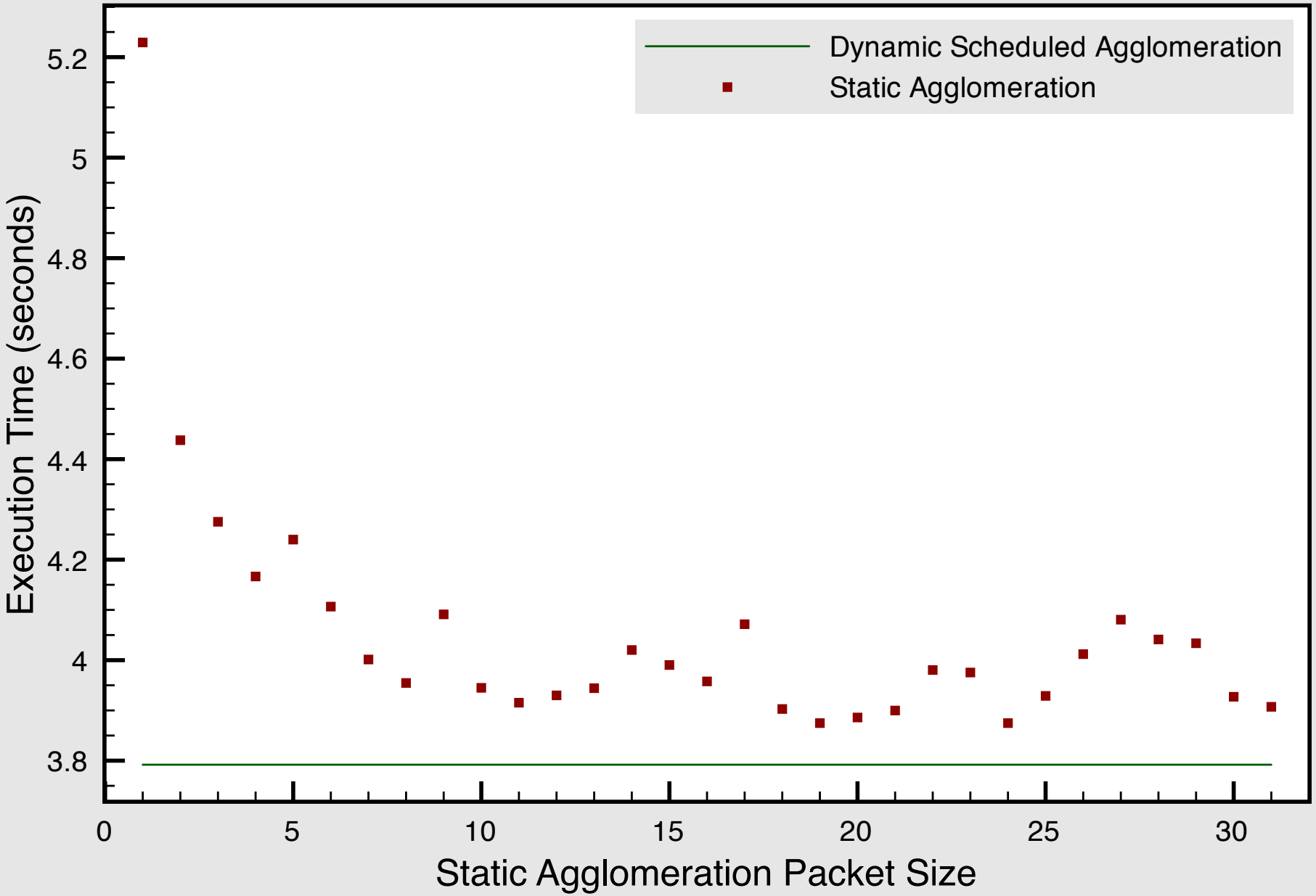
```
__global__ void interact(...) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  // For loop added for agglomeration
  for(int j = start[i]; j < end[i]; j++)
    // interaction work
}
```

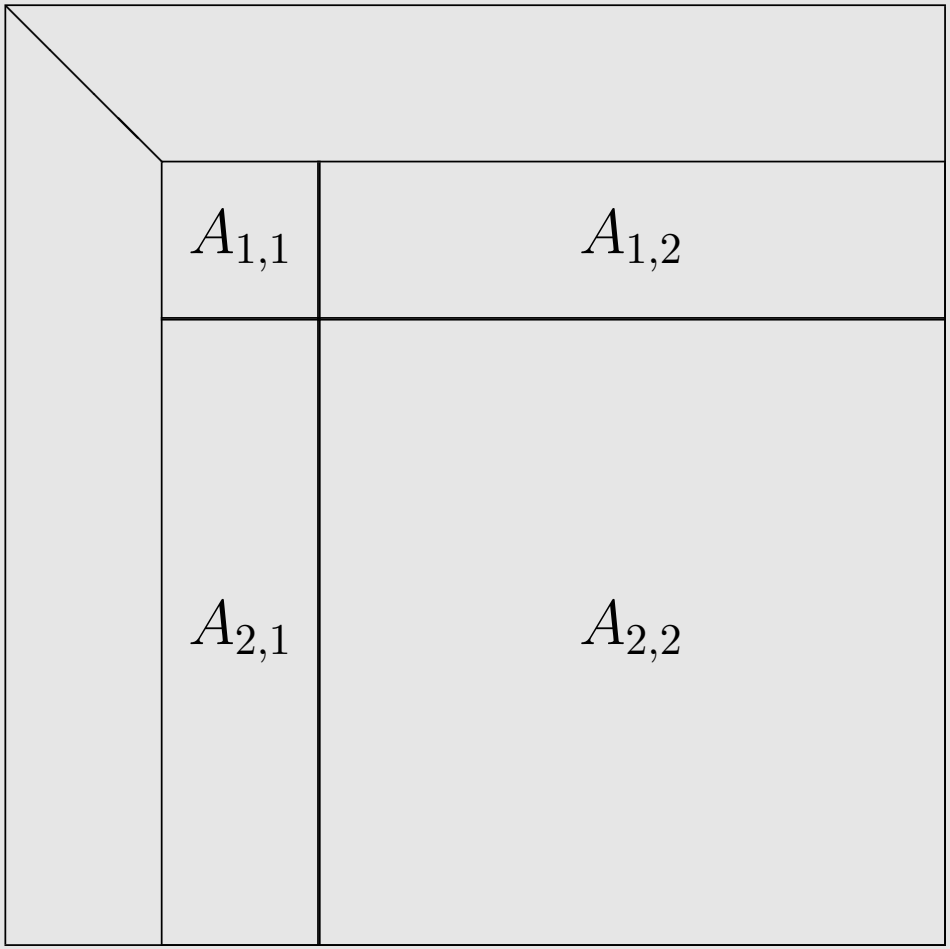

CPU only
GPU without agglomeration
GPU with agglomeration
Execution Time (seconds)
0
20
40
60
80
100
120
140
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Particles


Speedup of Agglomeration
1.18
1.15
1.12
1.09
1.06
1.03
1
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Particles


GPU without agglomeration
GPU with agglomeration
Execution Time (seconds)
120
115
110
105
100
95
90
85
500
1000
1500
2000
2500
Number of Particles per Work Unit

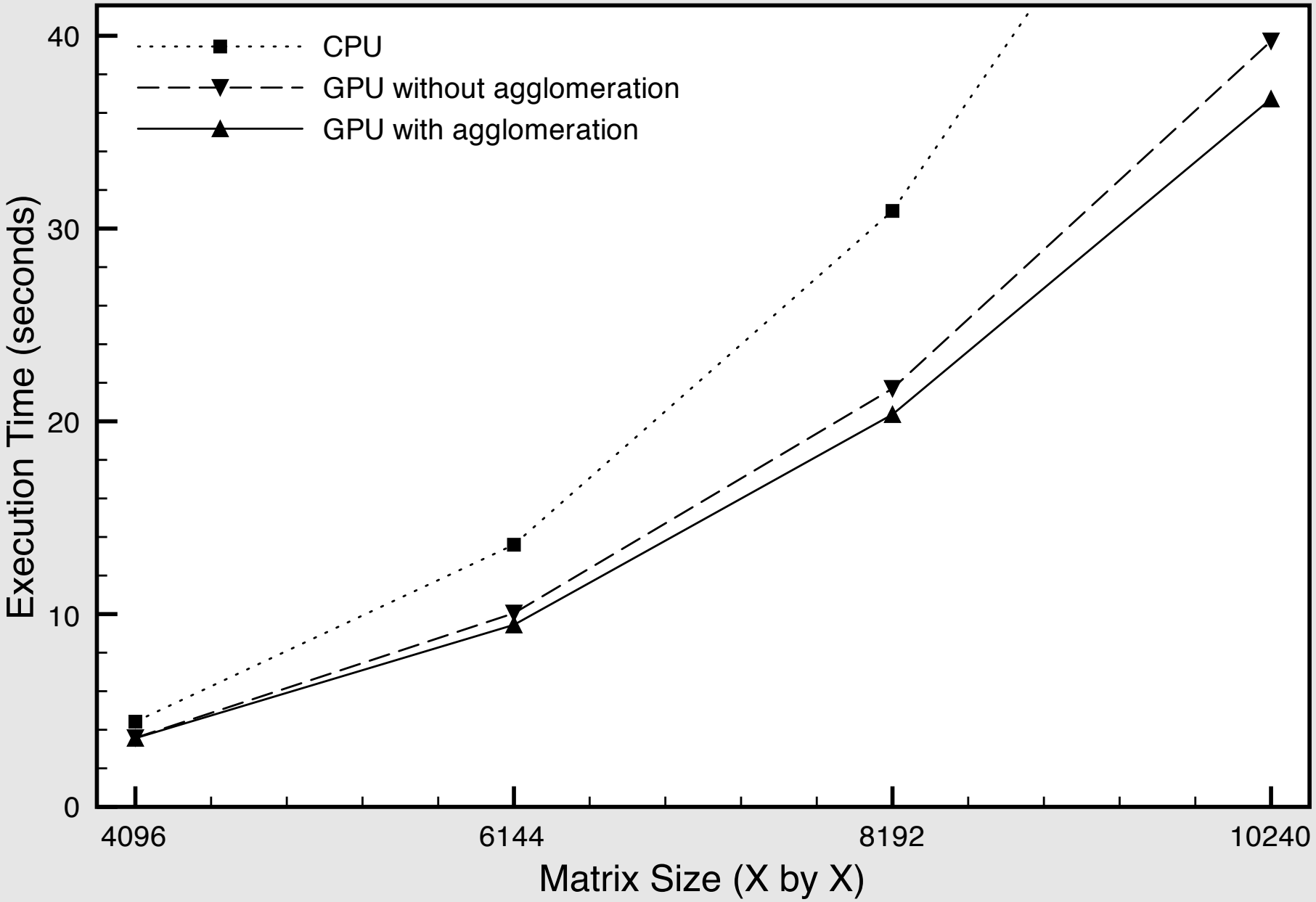

Dynamic Scheduled Agglomeration
Static Agglomeration
Execution Time (seconds)
5.2
5
4.8
4.6
4.4
4.2
4
3.8
0
5
10
15
20
25
30
Static Agglomeration Packet Size

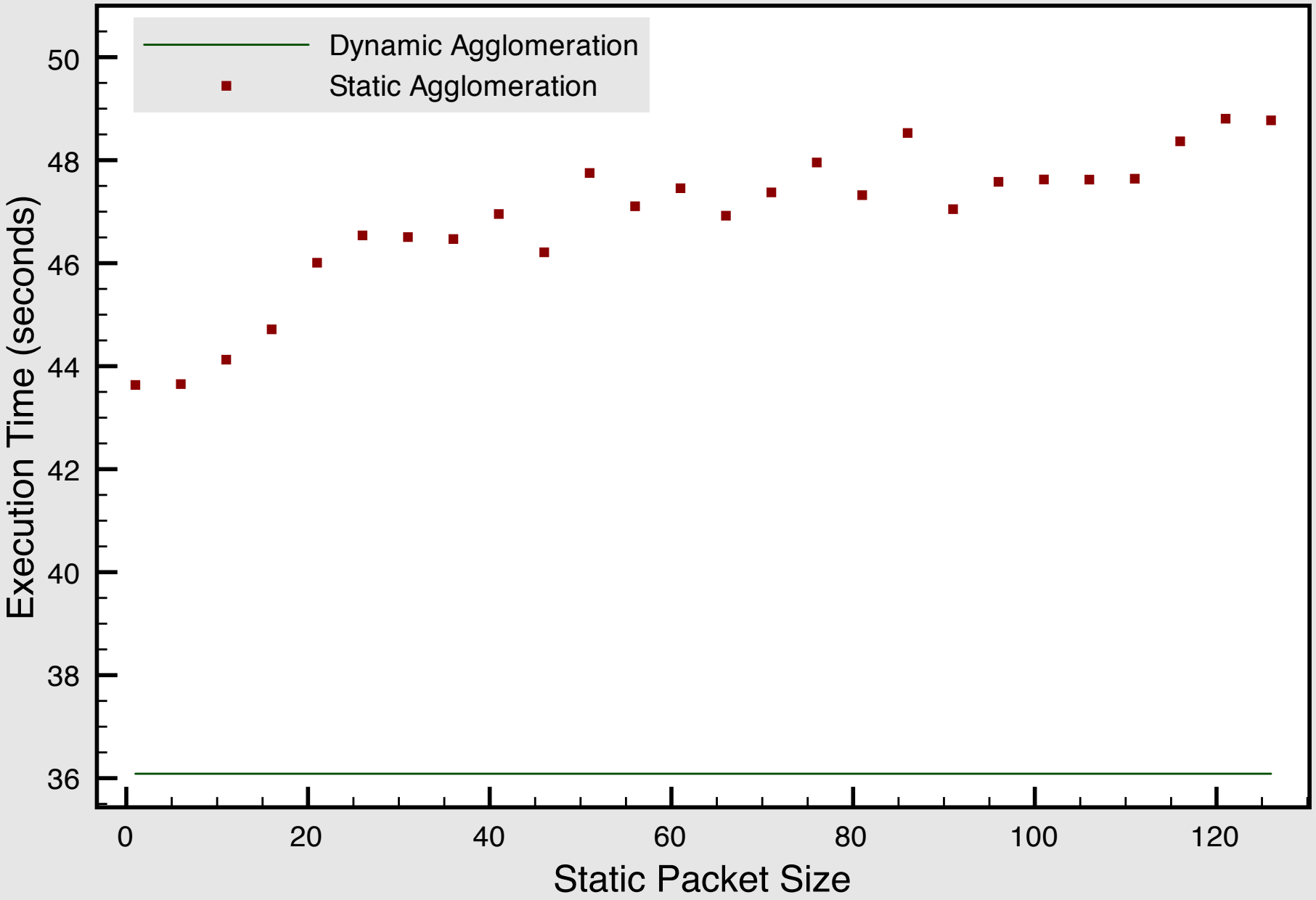
Application: LU Factorization without pivoting


$A_{1,1}$
$A_{1,2}$
$A_{2,1}$
$A_{2,2}$

LU Factorization

- CPU
  - Diagonal factorization
  - Triangular solves
- GPU
  - Matrix-matrix multiples (DGEMMs)


CPU
GPU without agglomeration
GPU with agglomeration
Execution Time (seconds)
0
10
20
30
40
4096
6144
8192
10240
Matrix Size (X by X)


Dynamic Agglomeration
Static Agglomeration
Execution Time (seconds)
50
48
46
44
42
40
38
36
0
20
40
60
80
100
120
Static Packet Size

Conclusion

- For both benchmarks, agglomerating work increases performance
- Agglomeration does not need to be application-specific
- Statically selecting work units to agglomerate is difficult and may reduce performance
- Runtimes can agglomerate automatically
  - An agglomerating kernel still must written
  - Obtains better performance than static