Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters

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

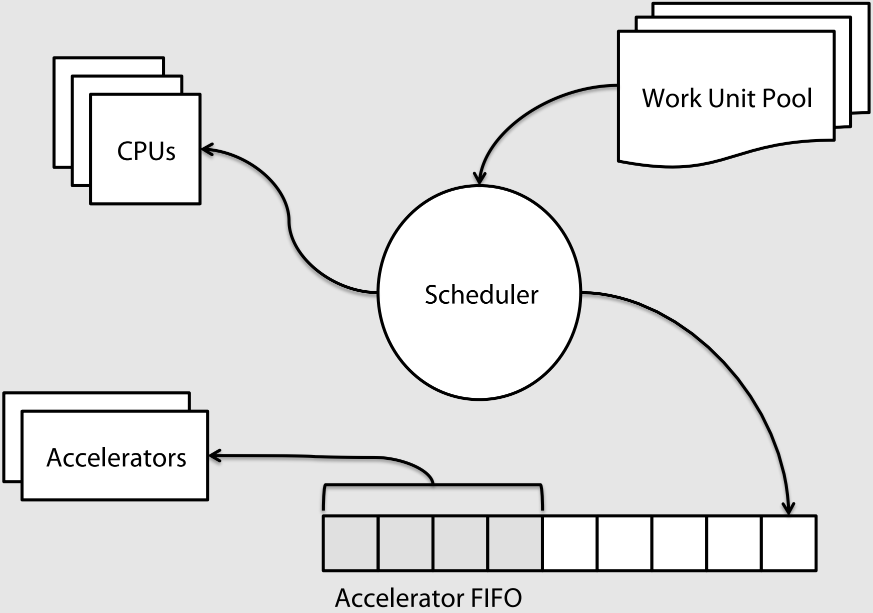
University of Illinois Urbana-Champaign

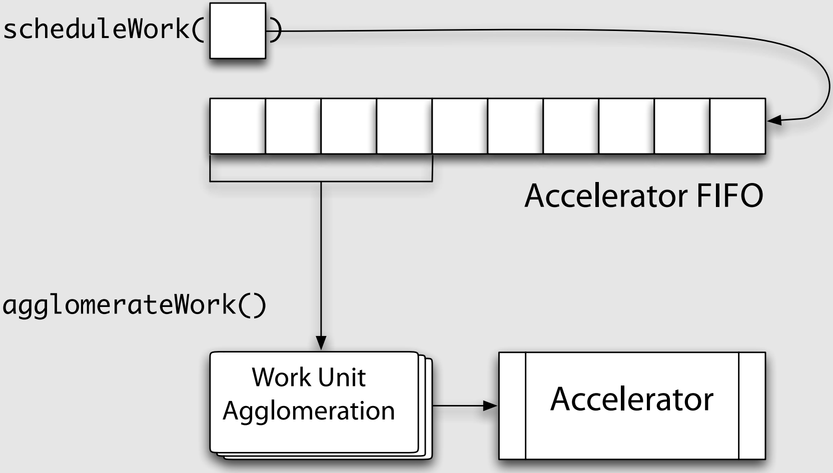
May 7, 2012

- Work is *overdecomposed* in objects
    - Fine-grain task parallelism
    - Ideal for 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

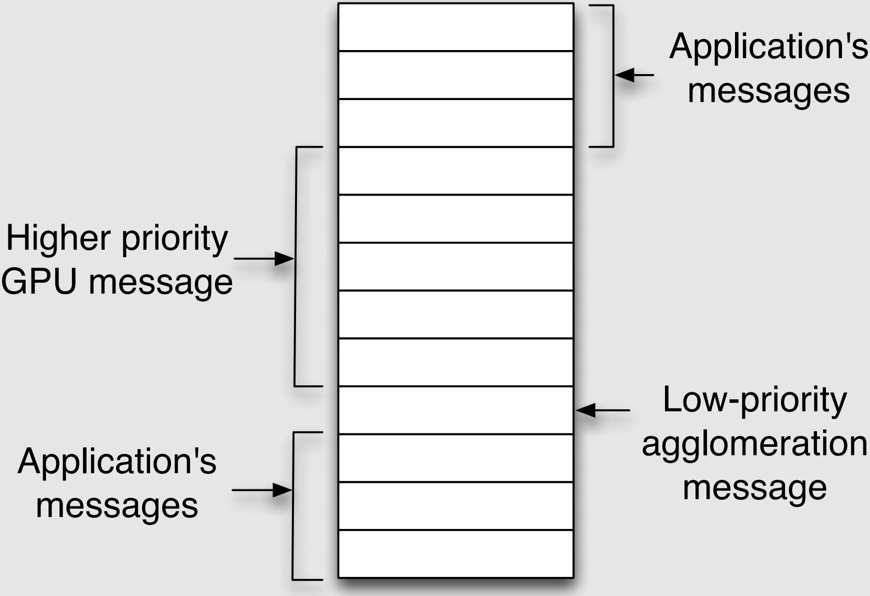

Work Unit Pool
CPUs
Scheduler
Accelerators
Accelerator FIFO



scheduleWork( )
Accelerator FIFO
agglomerateWork()
Work Unit Agglomeration
Accelerator

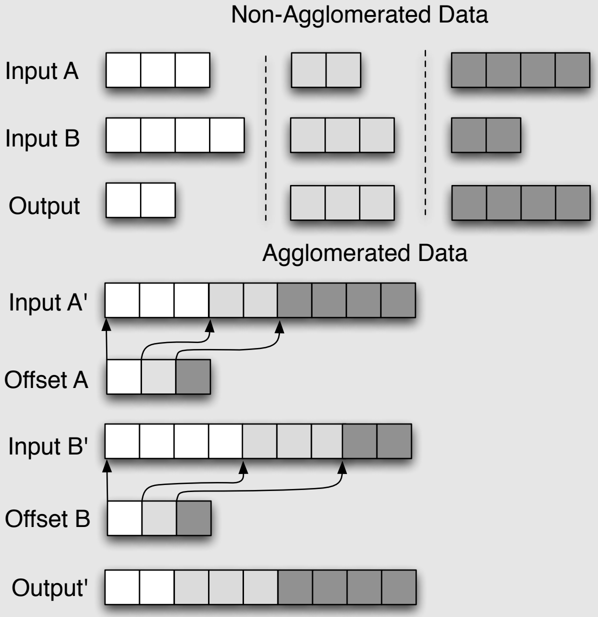
Programmer/Runtime Division

- 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


Application's messages
Higher priority GPU message
Low-priority agglomeration message
Application's messages

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 impact performance


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

Case study: Molecular2D

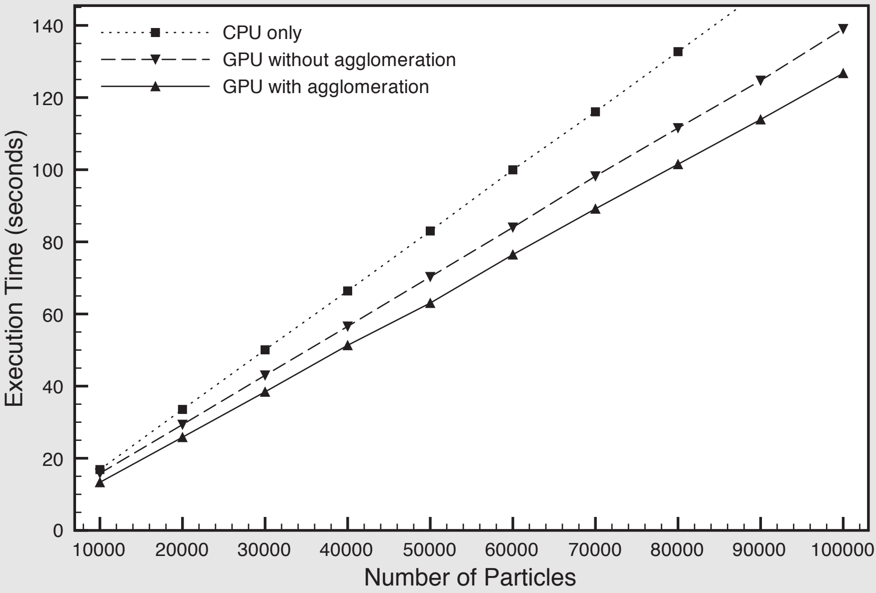
Molecular2D

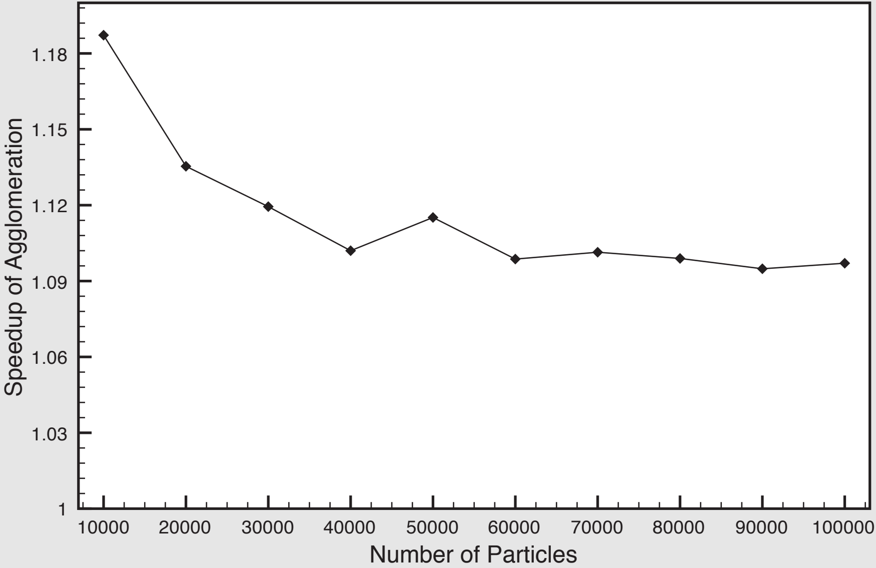
- Cells
  - Execute on CPU
- Interactions
  - Execute on GPU

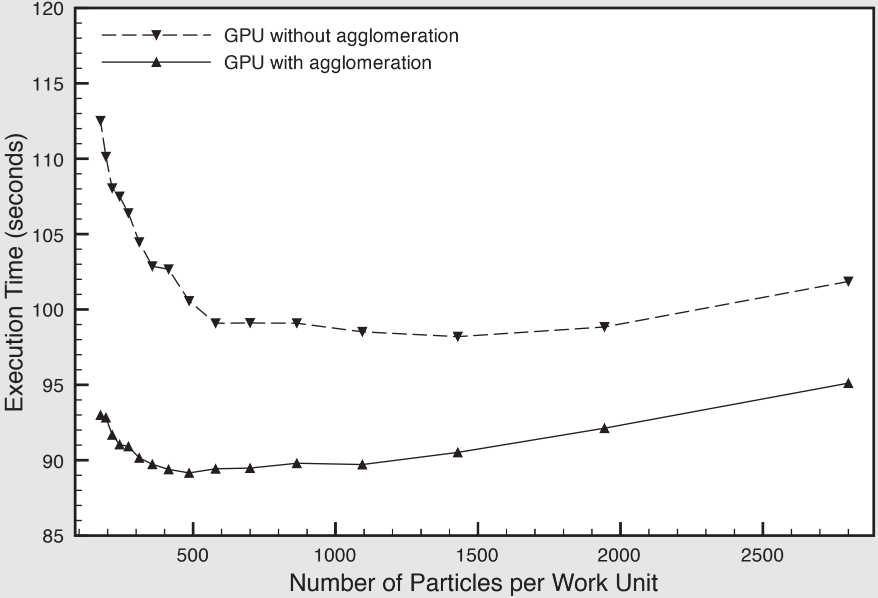
Molecular 2D Interaction Kernel

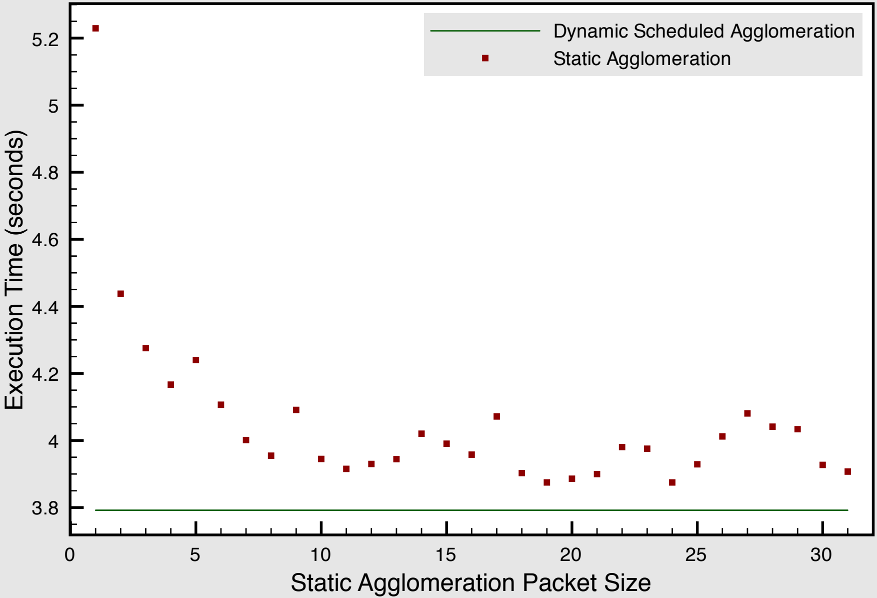
```
__global__ void interact(...) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

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

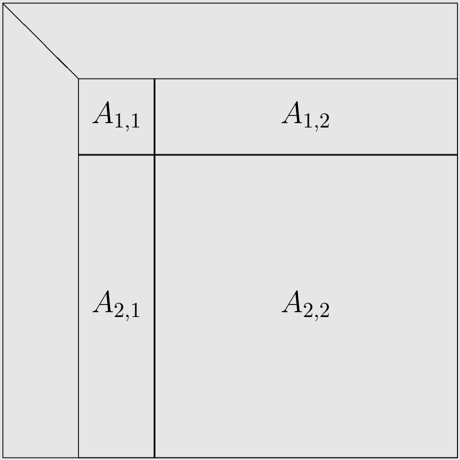

CPU only
GPU without agglomeration
GPU with agglomeration
Execution Time (seconds)
0
20
40
60
80
100
120
140
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Particles


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

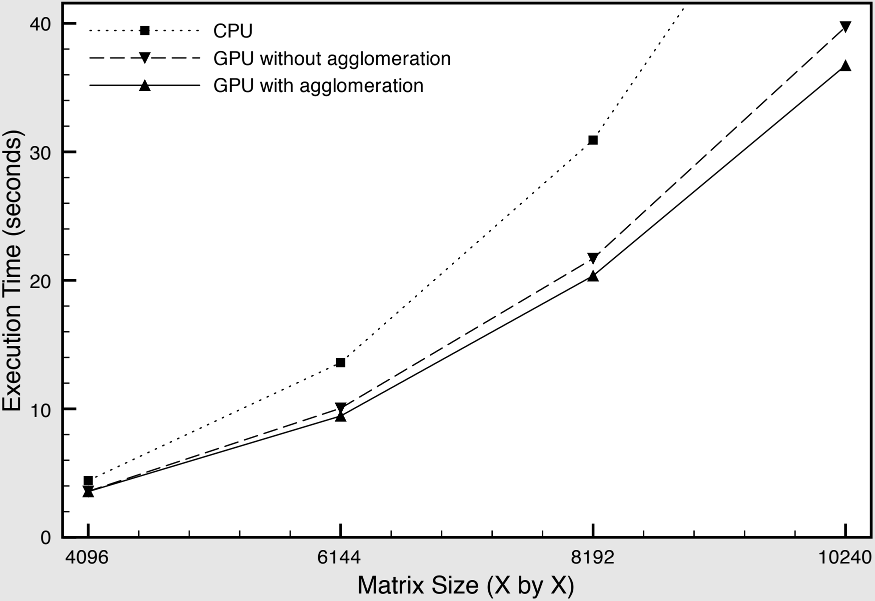

Dynamic Scheduled Agglomeration
Static Agglomeration
Execution Time (seconds)
Static Agglomeration Packet Size
5.2
5
4.8
4.6
4.4
4.2
4
3.8
0
5
10
15
20
25
30

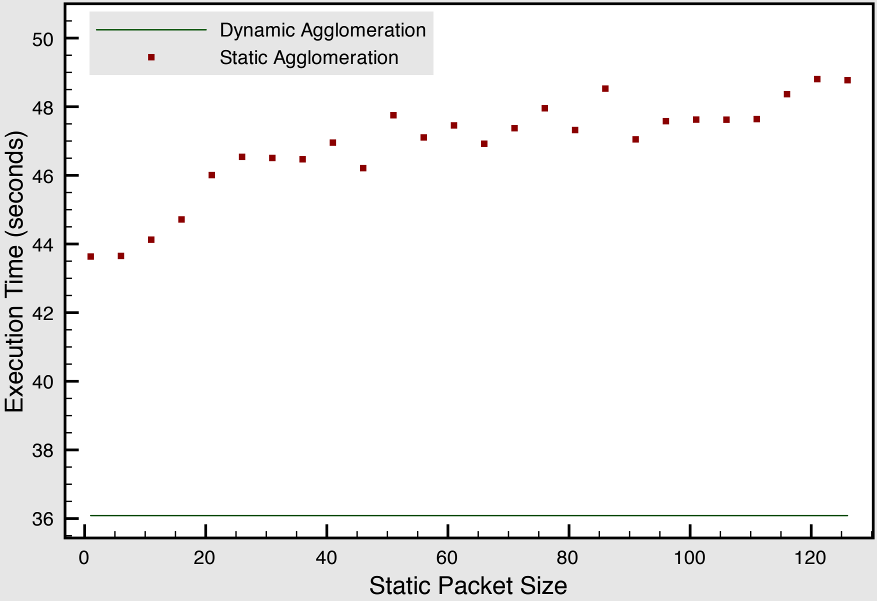
Case study: LU Factorization without pivoting


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

LU Factorization

- CPU
  - Diagonal
  - Triangular solves
- GPU
  - Matrix-matrix multiples


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