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

Roshan Dathathri    Chandan Reddy
Thejas Ramashekar   Uday Bondhugula

Department of Computer Science and Automation

Indian Institute of Science

{roshan,chandan.g,thejas,uday}@csa.iisc.ernet.in

September 11, 2013
OpenMP code for shared-memory systems:

```c
for (i = 1; i <= N; i++) {
    #pragma omp parallel for
    for (j = 1; j <= N; j++) {
        <computation>
    }
}
```

MPI code for distributed-memory systems:

```c
for (i = 1; i <= N; i++) {
    set_of_j_s = dist(1, N, processor_id);
    for each j in set_of_j_s {
        <computation>
    }
    <communication>
}
```

Explicit communication is required between:

- devices in a heterogeneous system with CPUs and multiple GPUs.
- nodes in a distributed-memory cluster.

Hence, tedious to program.
Affine loop nests

- Arbitrarily nested loops with affine bounds and affine accesses.
- Form the compute-intensive core of scientific computations like:
  - stencil style computations,
  - linear algebra kernels,
  - alternating direction implicit (ADI) integrations.
- Can be analyzed by the polyhedral model.
Example iteration space
Example iteration space
Example iteration space

- Dependence (1,0)
- Dependence (0,1)
- Dependence (1,1)

Tiles

Parallel phases

Multicore Computing Lab (CSA, IISc)
For affine loop nests:

- Statically determine data to be transferred between compute devices.
  - with a goal to move only those values that need to be moved to preserve program semantics.

- Generate data movement code that is:
  - parametric in problem size symbols and number of compute devices.
  - valid for any computation placement.
Communication is parameterized on a tile

- Tile represents an iteration of the innermost distributed loop.
- May or may not be the result of loop tiling.
- A tile is executed atomically by a compute device.
Existing flow-out (FO) scheme

Flow-out set:
- The values that need to be communicated to other tiles.
- Union of per-dependence flow-out sets of all RAW dependences.
Existing flow-out (FO) scheme

There are tiles that require the flow-out set. They are highlighted in green.
Existing flow-out (FO) scheme

- All elements in the flow-out set might not be required by all its receiving tiles.
- Only ensures that the receiver requires at least one element in the communicated set.
- Could transfer unnecessary elements.
Our first scheme

Motivation:

- All elements in the data communicated should be required by the receiver.

Key idea:

- Determine data that needs to be sent from one tile to another, parameterized on a sending tile and a receiving tile.
Flow-in (FI) set

Flow-in set:

- The values that need to be received from other tiles.
- Union of per-dependence flow-in sets of all RAW dependences.
Flow-out intersection flow-in (FOIFI) scheme

Flow set:
- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.
Flow-out intersection flow-in (FOIFl) scheme

Flow set:
- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.
Flow-out intersection flow-in (FOIFl) scheme

Flow set:

- Parameterized on two tiles.
- The values that need to be communicated from a sending tile to a receiving tile.
- Intersection of the flow-out set of the sending tile and the flow-in set of the receiving tile.
Flow-out intersection flow-in (FOIFI) scheme

- Precise communication when each receiving tile is executed by a different compute device.
- Could lead to huge duplication when multiple receiving tiles are executed by the same compute device.
Comparison with virtual processor based schemes

- Some existing schemes use a virtual processor to physical mapping to handle symbolic problem sizes and number of compute devices.
- Tiles can be considered as virtual processors in FOIFI.
- Lesser redundant communication in FOIFI than prior works that use virtual processors since it:
  - uses exact-dataflow information.
  - combines data to be moved due to multiple dependences.
Our main scheme

Motivation:
- Partitioning the communication set such that all elements within each partition is required by all receivers of that partition.

Key idea:
- Partition the dependences in a particular way, and determine communication sets and their receivers based on those partitions.
Flow-out partitioning (FOP) scheme

Source-distinct partitioning of dependences - partitions dependences such that:

- all dependences in a partition communicate the same set of values.
- any two dependences in different partitions communicate disjoint set of values.

Determine communication set and receiving tiles for each partition.
Communication sets of different partitions are disjoint.

Union of communication sets of all partitions yields the flow-out set.

Hence, the flow-out set of a tile is partitioned.
Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.
Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.
Initially, each dependence is:

- restricted to those constraints which are inter-tile, and
- put in a separate partition.
Source-distinct partitioning of dependences

For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
Source-distinct partitioning of dependences

For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
Source-distinct partitioning of dependences

For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
For all pairs of dependences in two partitions:

- Find the source iterations that access the same region of data - source-identical.
- Get new dependences by restricting the original dependences to the source-identical iterations.
- Subtract out the new dependences from the original dependences.

The set of new dependences formed is a new partition.
Source-distinct partitioning of dependences

Stop when no new partitions can be formed.
Flow-out partitioning (FOP) scheme: at runtime

For each partition and tile executed, one of these is chosen:

- **multicast-pack**: the partitioned communication set from this tile is copied to the buffer of its receivers.
- **unicast-pack**: the partitioned communication set from this tile to a receiving tile is copied to the buffer of that receiver.

**unicast-pack** is chosen only if each receiving tile is executed by a different receiver.
Flow-out partitioning (FOP) scheme

- Reduces granularity at which receivers are determined.
- Reduces granularity at which the conditions to choose between multicast-pack and unicast-pack are applied.
- Minimizes communication of both duplicate and unnecessary elements.
Another example - dependences

Let:

\((k, i, j)\) - source iteration
\((k', i', j')\) - target iteration

Dependence 1:

\[ k' = k + 1 \]
\[ i' = i \]
\[ j' = j \]

Dependence 2:

\[ k' = k + 1 \]
\[ i' = i \]
\[ j = k + 1 \]
Another example - FO scheme

![Diagram showing FO scheme with tiles and flow-out set]

Dependencies:
- Dependence 1
- Dependence 2

Tiles: $j = k + 1$
Another example - FOIFI scheme

Flow set
Tiles
Dependence 1
Dependence 2

i

k

j

j = k + 1
Another example - FOIFI scheme
Another example - FOIFI scheme

\[ j = k + 1 \]
Another example - FOP scheme
Another example - FOP scheme

Dependence1
Tiles
Flow-out partition

\[ j = k + 1 \]
As part of the PLUTO framework.

Input is sequential C code which is tiled and parallelized using the PLUTO algorithm.

Data movement code is automatically generated using our scheme.
Code for distributed-memory systems using existing techniques is automatically generated.

Asynchronous MPI primitives are used to communicate between nodes in a distributed-memory system.
For heterogeneous systems, the host CPU acts both as a compute device and as the orchestrator of data movement between compute devices, while the GPU acts only as a compute device.

OpenCL functions `clEnqueueReadBufferRect()` and `clEnqueueWriteBufferRect()` are used for data movement in heterogeneous systems.
32-node InfiniBand cluster.
Each node consists of two quad-core Intel Xeon E5430 2.66 GHz processors.
The cluster uses MVAPICH2-1.8 as the MPI implementation.
Benchmarks

- Floyd Warshall (floyd).
- LU Decomposition (lu).
- Alternating Direction Implicit solver (adi).
- 2-D Finite Different Time Domain Kernel (fdtd-2d).
- Heat 2D equation (heat-2d).
- Heat 3D equation (heat-3d).

The first 4 are from Polybench/C 3.2 suite, while heat-2d and heat-3d are widely used stencil computations.
Comparison of FOP, FOIFI and FO

- Same parallelizing transformation -> same frequency of communication.
- Differ only in the communication volume.
- Comparing execution times directly compares their efficiency.
Comparison of FOP with FO

- Communication volume reduced by a factor of $1.4\times$ to $63.5\times$.
- Communication volume reduction translates to significant speedup, except for heat-2d.
- Speedup of upto $15.9\times$.
- Mean speedup of $1.55\times$. 
Similar behavior for stencil-style codes.

For floyd and lu:
- Communication volume reduced by a factor of $1.5 \times$ to $31.8 \times$.
- Speedup of upto $1.84 \times$.

Mean speedup of $1.11 \times$. 
OMPD - OpenMP to MPI

- Takes OpenMP code as input and generates MPI code.
- Primarily a runtime dataflow analysis technique.
- Handles only those affine loop nests which have a repetitive communication pattern.
  - Communication should not vary based on the outer sequential loop.
- Cannot handle floyd, lu and time-tiled (outer sequential dimension tiled) stencil style codes.
Comparison of FOP with OMPD

- For heat-2d and heat-3d, significant speedup over OMPD.
  - The computation time is much lesser.
  - Better load balance and locality due to advanced transformations.
  - OMPD cannot handle such transformed code.

- For adi: significant speedup over OMPD.
  - Same volume of communication.
  - Better performance due to loop tiling.
  - Lesser runtime overhead.

- Mean speedup of $3.06 \times$. 
Unified Parallel C (UPC)

- Unified programming model for both shared-memory and distributed-memory systems.
- All benchmarks were manually ported to UPC.
  - Sharing data only if it may be accessed remotely.
  - UPC-specific optimizations like localized array accesses, block copy, one-sided communication.
For lu, heat-2d and heat-3d, significant speedup over UPC.
  • Better load balance and locality due to advanced transformations.
  • Difficult to manually write such transformed code.
  • UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.
Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.
- For adi: significant speedup over UPC.
  - Same computation time and communication volume.
  - Data to be communicated is not contiguous in memory.
  - UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.
Comparison of FOP with UPC

- For lu, heat-2d and heat-3d, significant speedup over UPC.
  - Better load balance and locality due to advanced transformations.
  - Difficult to manually write such transformed code.
  - UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.

- For adi: significant speedup over UPC.
  - Same computation time and communication volume.
  - Data to be communicated is not contiguous in memory.
  - UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.

- For fdtd-2d and floyd: UPC performs slightly better.
  - Same computation time and communication volume.
  - Data to be communicated is contiguous in memory.
  - UPC has no additional runtime overhead.
For lu, heat-2d and heat-3d, significant speedup over UPC.
- Better load balance and locality due to advanced transformations.
- Difficult to manually write such transformed code.
- UPC model is not suitable when the same data element could be written by different nodes in different parallel phases.

For adi: significant speedup over UPC.
- Same computation time and communication volume.
- Data to be communicated is not contiguous in memory.
- UPC incurs huge runtime overhead for such multiple shared memory requests to non-contiguous data.

For fdtd-2d and floyd: UPC performs slightly better.
- Same computation time and communication volume.
- Data to be communicated is contiguous in memory.
- UPC has no additional runtime overhead.

Mean speedup of $2.19 \times$. 
Results: distributed-memory cluster

Figure: FOP – strong scaling on distributed-memory cluster

Figure: floyd – speedup of FOP, FOIFI, FO and hand-optimized UPC code over seq on distributed-memory cluster

For the transformations and computation placement chosen:
FOP achieves the minimum communication volume.
Experimental evaluation: heterogeneous systems

Intel-NVIDIA system:

- Intel Xeon multicore server consisting of 12 Xeon E5645 cores.
- 4 NVIDIA Tesla C2050 graphics processors connected on the PCI express bus.
- NVIDIA driver version 304.64 supporting OpenCL 1.1.
Comparison of FOP with FO

- Communication volume reduced by a factor of $11 \times$ to $83 \times$.
- Communication volume reduction translates to significant speedup.
- Speedup of upto $3.47 \times$.
- Mean speedup of $1.53 \times$. 
Results: heterogeneous systems

Figure: FOP – strong scaling on the Intel-NVIDIA system

For the transformations and computation placement chosen:
FOP achieves the minimum communication volume.
Acknowledgements

AMD

Department of Science & Technology

Ministry of Science & Technology

NVIDIA

CUDA RESEARCH CENTER
Conclusions

- The framework we propose frees programmers from the burden of moving data.
- Partitioning of dependences enables precise determination of data to be moved.
- Our tool is the first one to parallelize affine loop nests for a combination of CPUs and GPUs while providing precision of data movement at the granularity of array elements.
- Our techniques will be able to provide OpenMP-like programmer productivity for distributed-memory and heterogeneous architectures if implemented in compilers.

Publicly available: http://pluto-compiler.sourceforge.net/
Results: distributed-memory cluster

### TABLE I: Total communication volume on distributed-memory cluster – FO and FOIFI normalized to FOP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem sizes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>floyd</td>
<td>8192^2</td>
<td>1.51GB</td>
<td>31.8×</td>
<td>3.53GB</td>
<td>15.9×</td>
</tr>
<tr>
<td></td>
<td>64^2</td>
<td>63.5×</td>
<td>FO</td>
<td>63.5×</td>
<td>FO</td>
</tr>
<tr>
<td></td>
<td>3.06x</td>
<td>FOIFI</td>
<td>15.9×</td>
<td>63.5×</td>
<td>FO</td>
</tr>
<tr>
<td>lu</td>
<td>4096^2</td>
<td>0.45GB</td>
<td>5.3×</td>
<td>0.99GB</td>
<td>3.0×</td>
</tr>
<tr>
<td></td>
<td>64^2</td>
<td>1.4×</td>
<td>FO</td>
<td>1.4×</td>
<td>FO</td>
</tr>
<tr>
<td></td>
<td>14.3×</td>
<td>FOIFI</td>
<td>3.0×</td>
<td>1.4×</td>
<td>FO</td>
</tr>
<tr>
<td>fftd-2d</td>
<td>1024x4096^2</td>
<td>0.21GB</td>
<td>1.0×</td>
<td>0.47GB</td>
<td>1.0×</td>
</tr>
<tr>
<td></td>
<td>16^2</td>
<td>14.3×</td>
<td>FO</td>
<td>15.1×</td>
<td>FO</td>
</tr>
<tr>
<td></td>
<td>1.0×</td>
<td>FOIFI</td>
<td>1.0×</td>
<td>15.1×</td>
<td>FO</td>
</tr>
<tr>
<td>heat-2d</td>
<td>1024x8192^2</td>
<td>0.75GB</td>
<td>2.0×</td>
<td>1.74GB</td>
<td>2.0×</td>
</tr>
<tr>
<td></td>
<td>256^3</td>
<td>2.0×</td>
<td>FO</td>
<td>2.0×</td>
<td>FO</td>
</tr>
<tr>
<td></td>
<td>5.61GB</td>
<td>2.0×</td>
<td>FOIFI</td>
<td>2.0×</td>
<td>FO</td>
</tr>
<tr>
<td>heat-3d</td>
<td>256x512^3</td>
<td>1.0×</td>
<td>2.0×</td>
<td>13.09GB</td>
<td>1.0×</td>
</tr>
<tr>
<td></td>
<td>16^4</td>
<td>2.0×</td>
<td>FO</td>
<td>2.0×</td>
<td>FO</td>
</tr>
<tr>
<td>adi</td>
<td>128x8192^2</td>
<td>191.24GB</td>
<td>4.0×</td>
<td>223.11GB</td>
<td>8.0×</td>
</tr>
<tr>
<td></td>
<td>256^3</td>
<td>4.0×</td>
<td>FO</td>
<td>8.0×</td>
<td>FO</td>
</tr>
</tbody>
</table>

### TABLE II: Total execution time on distributed-memory cluster – FOIFI, FO, OMPD and UPC normalized to FOP

(a) floyd – seq time is 2012s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2065.2s</td>
<td>1.01×</td>
<td>1.00×</td>
<td>0.98×</td>
</tr>
<tr>
<td>4</td>
<td>521.4s</td>
<td>1.10×</td>
<td>1.20×</td>
<td>0.97×</td>
</tr>
<tr>
<td>8</td>
<td>263.9s</td>
<td>1.18×</td>
<td>1.75×</td>
<td>0.97×</td>
</tr>
<tr>
<td>16</td>
<td>137.6s</td>
<td>1.33×</td>
<td>3.93×</td>
<td>0.97×</td>
</tr>
<tr>
<td>32</td>
<td>81.1s</td>
<td>1.46×</td>
<td>11.18×</td>
<td>0.93×</td>
</tr>
</tbody>
</table>

(b) lu – seq time is 82.9s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.5s</td>
<td>2.00×</td>
<td>1.00×</td>
<td>2.58×</td>
</tr>
<tr>
<td>4</td>
<td>9.1s</td>
<td>1.42×</td>
<td>1.02×</td>
<td>2.42×</td>
</tr>
<tr>
<td>8</td>
<td>5.4s</td>
<td>1.70×</td>
<td>1.05×</td>
<td>2.30×</td>
</tr>
<tr>
<td>16</td>
<td>4.1s</td>
<td>1.84×</td>
<td>1.05×</td>
<td>1.50×</td>
</tr>
<tr>
<td>32</td>
<td>3.9s</td>
<td>1.58×</td>
<td>1.00×</td>
<td>1.25×</td>
</tr>
</tbody>
</table>

(c) fftd-2d – seq time is 351.7s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>359.5s</td>
<td>1.00×</td>
<td>1.00×</td>
<td>0.98×</td>
</tr>
<tr>
<td>4</td>
<td>90.8s</td>
<td>1.00×</td>
<td>1.03×</td>
<td>1.26×</td>
</tr>
<tr>
<td>8</td>
<td>66.9s</td>
<td>1.00×</td>
<td>1.04×</td>
<td>1.01×</td>
</tr>
<tr>
<td>16</td>
<td>33.8s</td>
<td>1.00×</td>
<td>1.09×</td>
<td>1.01×</td>
</tr>
<tr>
<td>32</td>
<td>16.8s</td>
<td>1.00×</td>
<td>1.24×</td>
<td>0.99×</td>
</tr>
</tbody>
</table>

(d) heat-2d – seq time is 796.4s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>OMPD</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>228.3s</td>
<td>1.00×</td>
<td>1.00×</td>
<td>3.42×</td>
<td>5.33×</td>
</tr>
<tr>
<td>4</td>
<td>59.8s</td>
<td>1.00×</td>
<td>1.01×</td>
<td>3.29×</td>
<td>5.11×</td>
</tr>
<tr>
<td>8</td>
<td>31.4s</td>
<td>1.00×</td>
<td>1.02×</td>
<td>3.92×</td>
<td>5.47×</td>
</tr>
<tr>
<td>16</td>
<td>17.3s</td>
<td>1.00×</td>
<td>1.03×</td>
<td>3.58×</td>
<td>5.00×</td>
</tr>
<tr>
<td>32</td>
<td>10.2s</td>
<td>1.00×</td>
<td>1.04×</td>
<td>3.06×</td>
<td>4.25×</td>
</tr>
</tbody>
</table>

(e) heat-3d – seq time is 590.6s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>OMPD</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>235.5s</td>
<td>1.00×</td>
<td>1.00×</td>
<td>2.51×</td>
<td>2.68×</td>
</tr>
<tr>
<td>4</td>
<td>65.4s</td>
<td>1.00×</td>
<td>1.05×</td>
<td>2.39×</td>
<td>2.46×</td>
</tr>
<tr>
<td>8</td>
<td>36.1s</td>
<td>1.00×</td>
<td>1.15×</td>
<td>2.82×</td>
<td>2.54×</td>
</tr>
<tr>
<td>16</td>
<td>21.4s</td>
<td>1.00×</td>
<td>1.23×</td>
<td>2.58×</td>
<td>2.21×</td>
</tr>
<tr>
<td>32</td>
<td>14.1s</td>
<td>1.00×</td>
<td>1.33×</td>
<td>2.29×</td>
<td>1.78×</td>
</tr>
</tbody>
</table>

(f) adi – seq time is 2717s

<table>
<thead>
<tr>
<th>Nodes</th>
<th>FOP</th>
<th>FOIFI</th>
<th>FO</th>
<th>OMPD</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>422.7s</td>
<td>1.00×</td>
<td>0.95×</td>
<td>6.27×</td>
<td>7.90×</td>
</tr>
<tr>
<td>4</td>
<td>231.7s</td>
<td>1.00×</td>
<td>2.11×</td>
<td>3.55×</td>
<td>4.68×</td>
</tr>
<tr>
<td>8</td>
<td>143.6s</td>
<td>1.00×</td>
<td>4.00×</td>
<td>3.43×</td>
<td>4.29×</td>
</tr>
<tr>
<td>16</td>
<td>78.6s</td>
<td>1.00×</td>
<td>7.87×</td>
<td>2.88×</td>
<td>4.47×</td>
</tr>
<tr>
<td>32</td>
<td>41.0s</td>
<td>1.00×</td>
<td>15.9×</td>
<td>2.95×</td>
<td>5.22×</td>
</tr>
</tbody>
</table>

- Mean speedup of FOP over FO is 1.55x
- Mean speedup of FOP over OMPD is 3.06x
- Mean speedup of FOP over UPC is 2.19x
### Results: heterogeneous systems

**TABLE III: Results on the Intel-NVIDIA system**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem sizes</th>
<th>Tile sizes</th>
<th>Device combination</th>
<th>Total execution time</th>
<th>Total communication volume</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOP</td>
<td>FO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Speedup</td>
<td>Reduction</td>
</tr>
<tr>
<td>floyd</td>
<td>10240x10240</td>
<td>32x32</td>
<td>1 CPU (12 cores)</td>
<td>890s</td>
<td>148s 180s 1.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>113s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>- 43s 107s 2.49</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>- 65s 104s 1.60</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>lu</td>
<td>11264x11264</td>
<td>256x256</td>
<td>1 CPU (12 cores)</td>
<td>412s</td>
<td>92s 132s 1.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>77s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ftdt-2d</td>
<td>4096x10240x10240</td>
<td>32x32</td>
<td>1 CPU (12 cores)</td>
<td>1915s</td>
<td>580s 603s 1.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>397s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>heat-2d</td>
<td>4096x10240x10240</td>
<td>32x32</td>
<td>1 CPU (12 cores)</td>
<td>1112s</td>
<td>242s 255s 1.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>266s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>heat-3d</td>
<td>4096x512x512x512</td>
<td>32x32x32</td>
<td>1 CPU (12 cores)</td>
<td>3080s</td>
<td>1718s 2018s 1.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>1932s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 GPUs</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Mean speedup of FOP over FO is 1.53x
## TABLE IV: Results on the AMD system

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem sizes</th>
<th>Tile sizes</th>
<th>Device combination</th>
<th>Total execution time</th>
<th>Total communication volume</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOP</td>
<td>FO</td>
</tr>
<tr>
<td>floyd</td>
<td>10240x10240</td>
<td>32x32</td>
<td>1 CPU (4 cores)</td>
<td>1084s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>512s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>286s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdtd-2d</td>
<td>4096x5120x5120</td>
<td>32x32</td>
<td>1 CPU (4 cores)</td>
<td>1529s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>241s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>133s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heat-2d</td>
<td>4096x8192x8192</td>
<td>32x32</td>
<td>1 CPU (4 cores)</td>
<td>3654s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>256s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 GPUs</td>
<td>-</td>
<td>142s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>