HyperFlow: An Efficient Dataflow Architecture for Multi CPU-GPU Systems

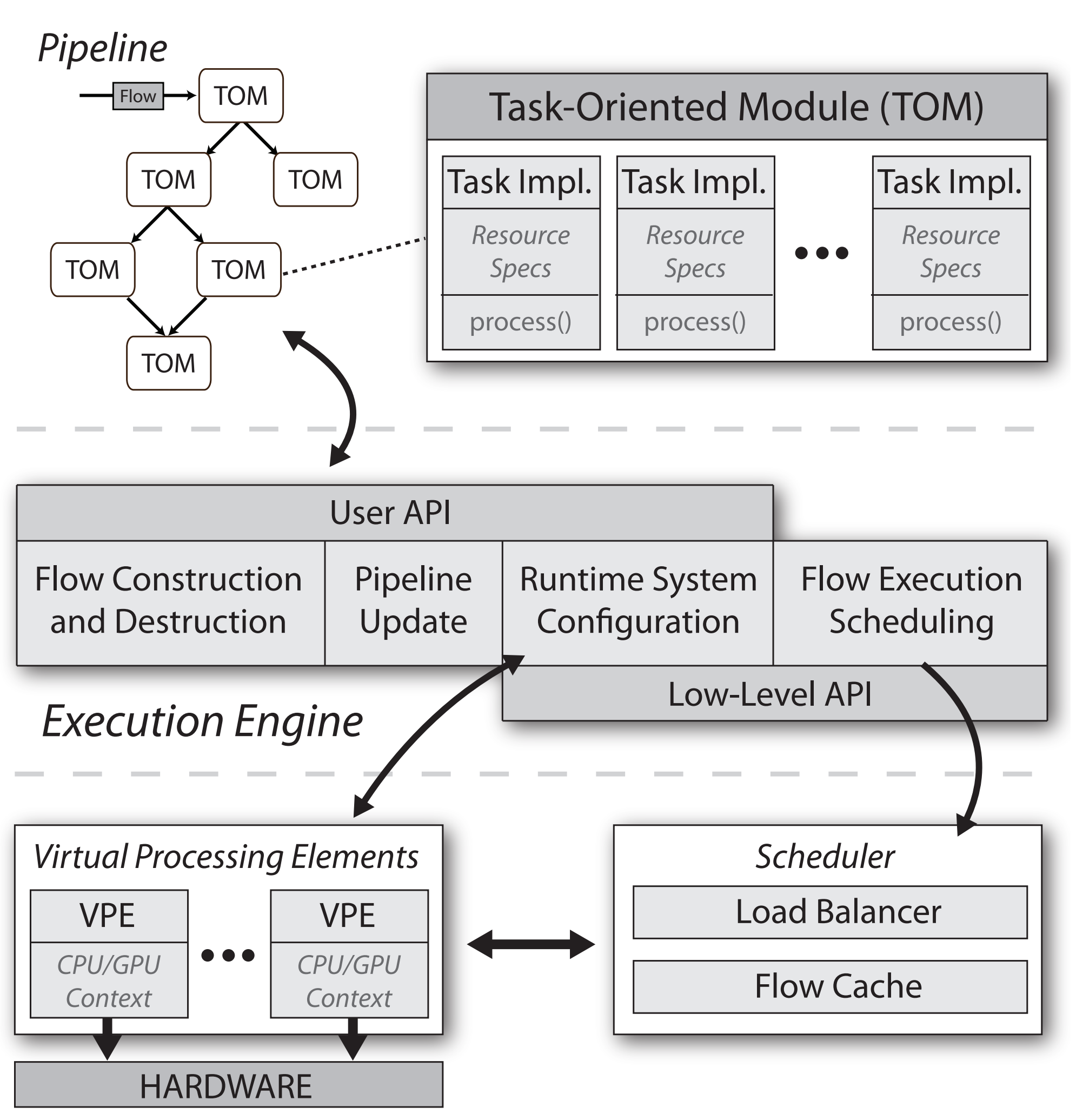
Huy Vo[1], Daniel Osmari[2], Luiz Scheidegger[2], João Comba[2], Jason Shepherd[3] and Cláudio Silva[1]

[1] University of Utah, USA [2] UFRGS, Brazil [3] Sandia National Laboratories, USA

WHY HYPERFLOW?

- Dataflow is widely used in scientific computing
- Current dataflow systems lack parallelism support:
  - Mostly designed for single-core machines
  - Some only support data-parallelism with MPI
  - Ad-hoc for multi-core and (multi-)GPU platforms
- There exists parallel frameworks such as TBB, StarPU Stream SDK, etc. but difficult to apply to dataflow due to being tightly coupled with task-graph execution.

SYSTEM ARCHITECTURE



TASK-ORIENTED MODULE (TOM)

- Hold meta-data for its task and parameters
- Manage multiple implementations for a specific task
- Each implementation specifies its own resources

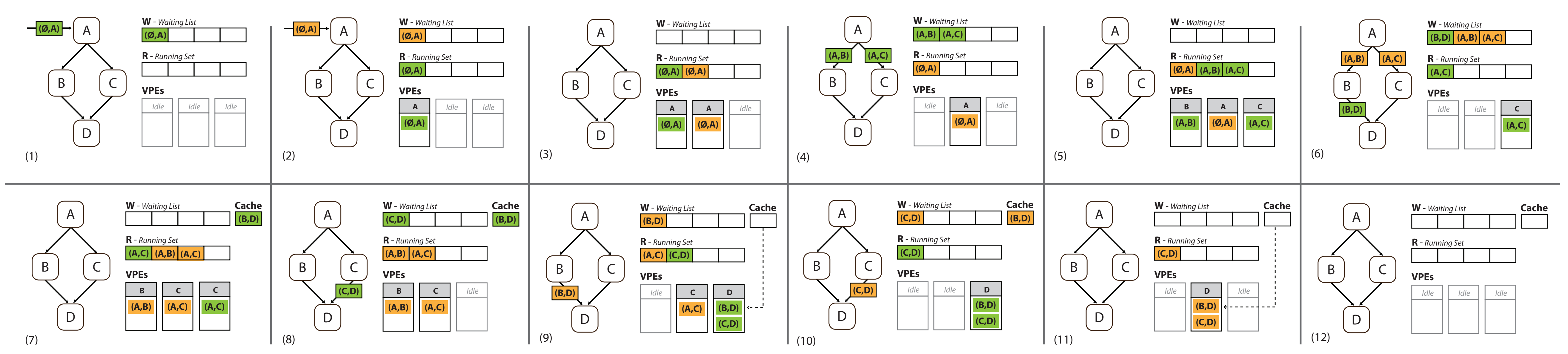
```cpp
class TaskOrientedModule {
public:
  TaskOrientedModule(int numberOfInputPorts,
                     int numberOfOutputPorts);
  void addImplementation(TaskImplementation* impl);
  static void connect(
      TaskOrientedModule* srcModule, int srcPort,
      TaskOrientedModule* dstModule, int dstPort
    );
  // Additional parameters if subclassed
  ...
};

// Construction of the pipeline
TaskOrientedModule Source(0, 1, "Image Reader")
Source.addImplementation(new SourceImpl());

TaskOrientedModule Filter(1, 1, "Gaussian Blur")
Filter.addImplementation(new FilterImpl());

TaskOrientedModule Sink(1, 0, "Viewer")
Sink.addImplementation(new SinkImpl());

TaskOrientedModule::connect(&Source,0,&Filter,0);
TaskOrientedModule::connect(&Filter,0,&Sink,0);
```

MULTI-MEDIUM DATA INTERFACE

- Data explicitly denotes which platform to reside in
- Data transferring is triggered automatically but conversions are done by user-define functions

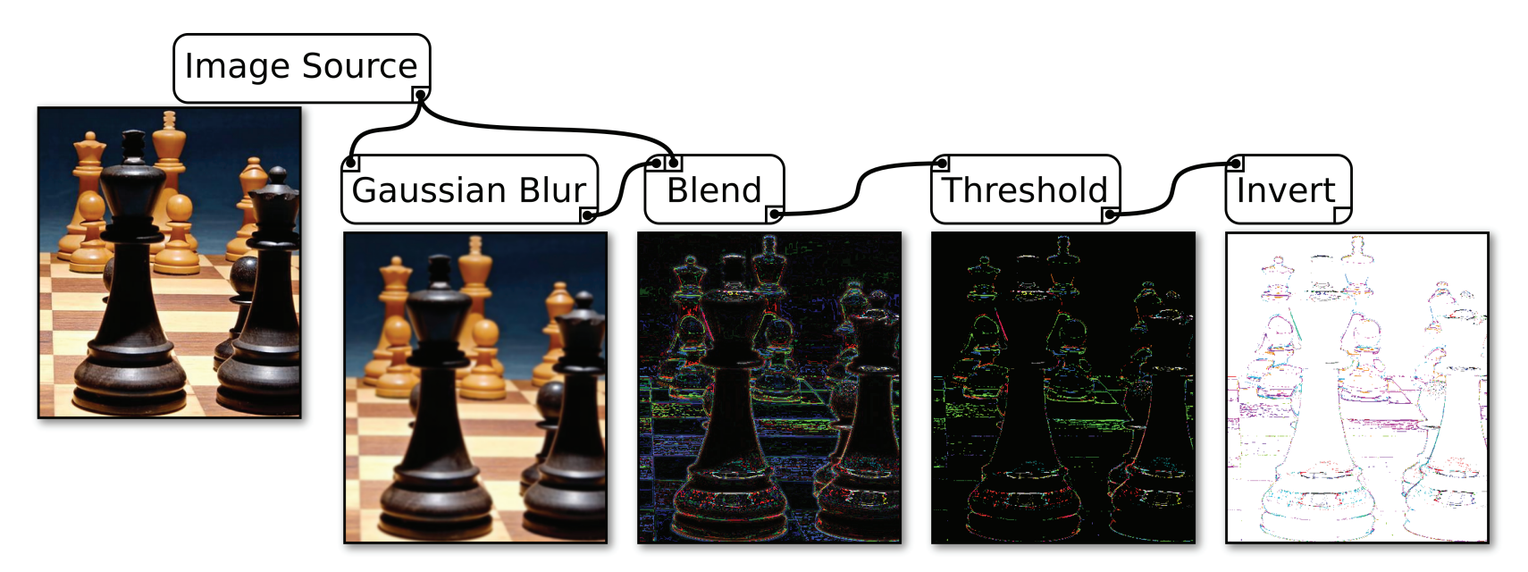
```cpp
class DImageData : public Data {
public:
  DImageData(BaseImage *img, DMTYPE medium) {
    // Store image with the medium type
    // ...
  }
  virtual Data* createInstance(DMTYPE targetMedium) {
    // Convert from CPU Image to CUDA Image
    if (this->medium==DM_CPU_MEMORY &&
        targetMedium==DM_GPU_MEMORY) {
      CUDAImage *newImage = new CUDAImage();
      cudaMemcpy(newImage, this->image,
           cudaMemcpyHostToDevice);
      return new DImageData(newImage, DM_GPU_MEMORY);
    }
    // Convert from CUDA Image to CPU Image
    else if (this->medium==DM_GPU_MEMORY &&
             targetMedium==DM_CPU_MEMORY) {
      CPUImage *newImage = new CPUImage();
      cudaMemcpy(newImage, this->image,
           cudaMemcpyDeviceToHost);
      return new DImageData(newImage, DM_CPU_MEMORY);
    }
    // Use default transfer methods for CPU-CPU, etc.
    else
      return Data::createInstance(medium);
  }
```

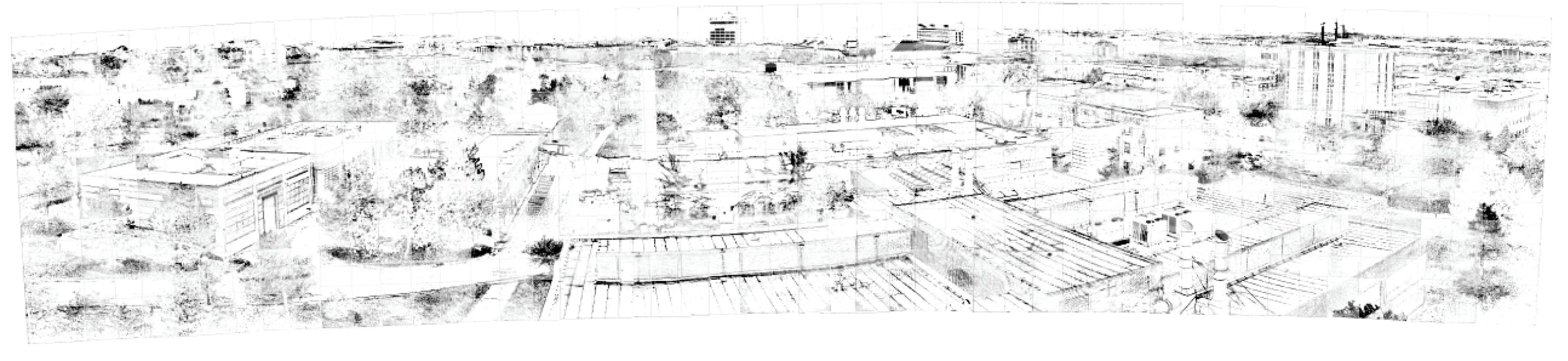
PIPELINE EXECUTION



SHOW CASE 1 - HIGH-THROUGHPUT IMAGE PROCESSING

Edge Detection Pipeline





Machine Specs:
(1) 2 x Xeon w5580, 8-core
24GB RAM
Quadro 5800
(2) Intel 970i, 4-core
6GB RAM
GTX 295, Tesla C1060

Timing Results
Input: 312 images ~ 6GB

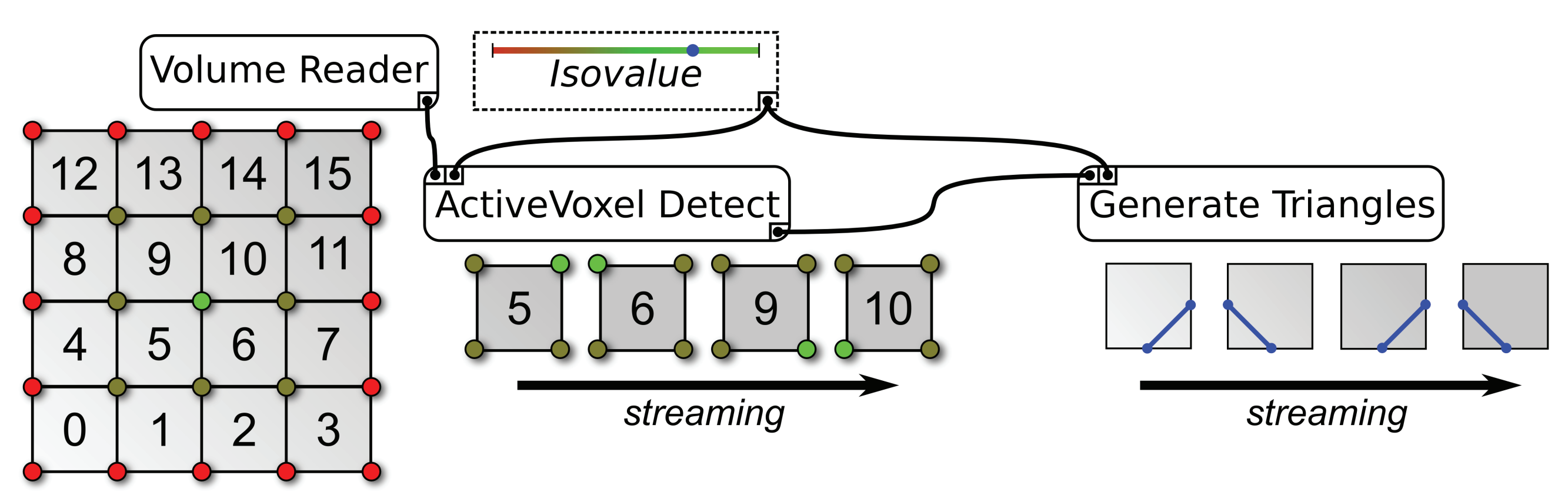
#CPU Cores	#GPU Cores	Time (s)
1	0	3261
2	0	1747
4	0	1049
8	0	668
8	1	440
8	2	272
4	3	151

SHOW CASE 2 - HYBRID ISOSURFACE EXTRACTION

The pipeline starts with a dataset reader that loads blocks of the volume and stream them to a TOM classifier. The classifier module traverses each block to generate a list of active voxels, i.e., only voxels that may contain the desired isosurface (voxels that do not contain v can be safely discarded). The list of active voxels is progressively streamed to a triangle generation TOM, which is designed to inspect each voxel and, based on a standard Marching Cubes case table, construct a set of triangles that approximates the desired isosurface inside that voxel. This set of triangles can also be streamed to other modules for further post-processing.

Using HyperFlow, it becomes very easy to optimize the usage of CPU and GPU computational resources. We implemented the dataset reader to divide the input volume into blocks. The execution scheduler automatically instantiates separate, concurrent TOMs to perform voxel classification on each block. Since there is no data dependency between different instances, this parallelism greatly improves algorithm performance. The TOMs designed to construct the actual isosurface can also be trivially parallelized by HyperFlow.

Isosurface Extraction Pipeline

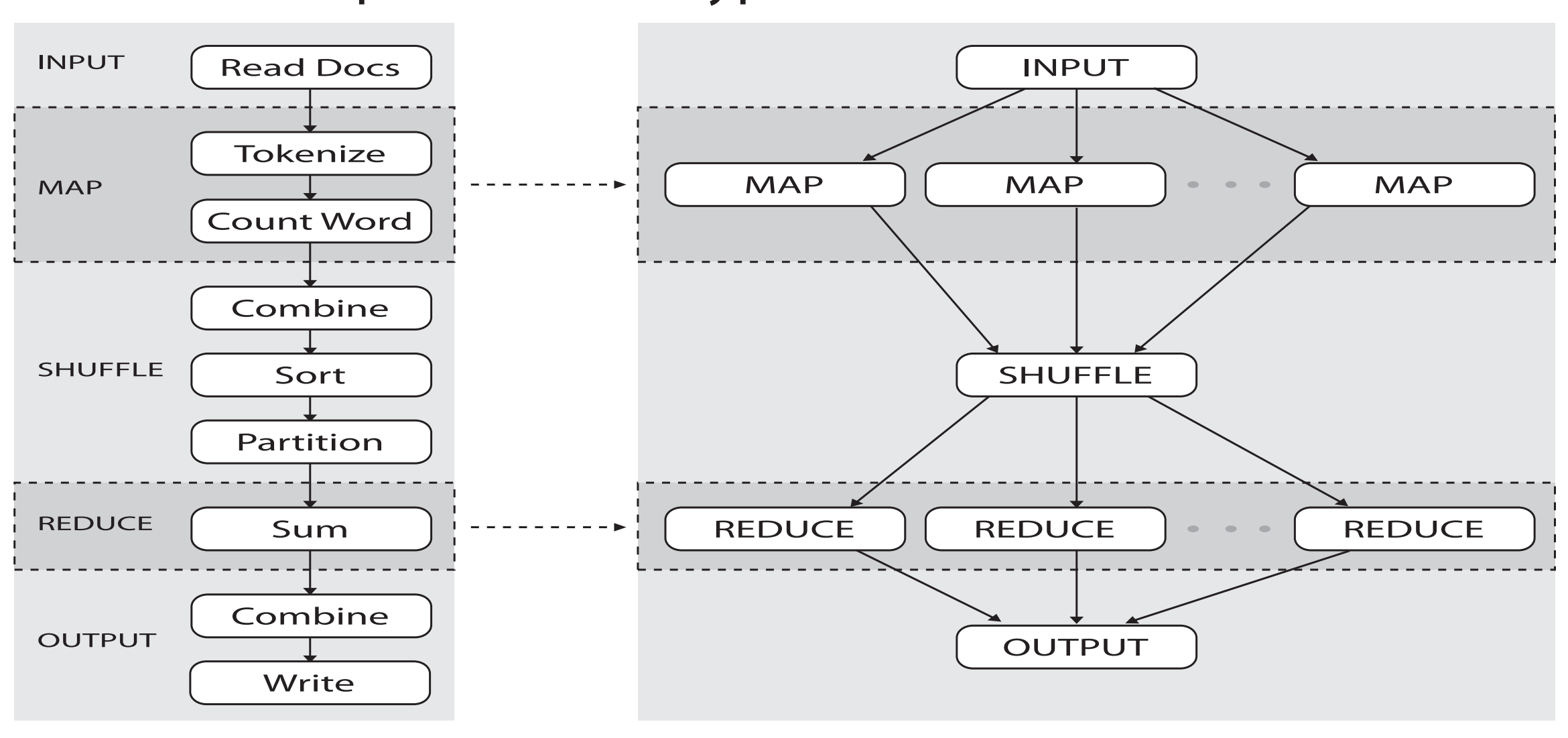


Timing Results
Input: 1 billion voxels

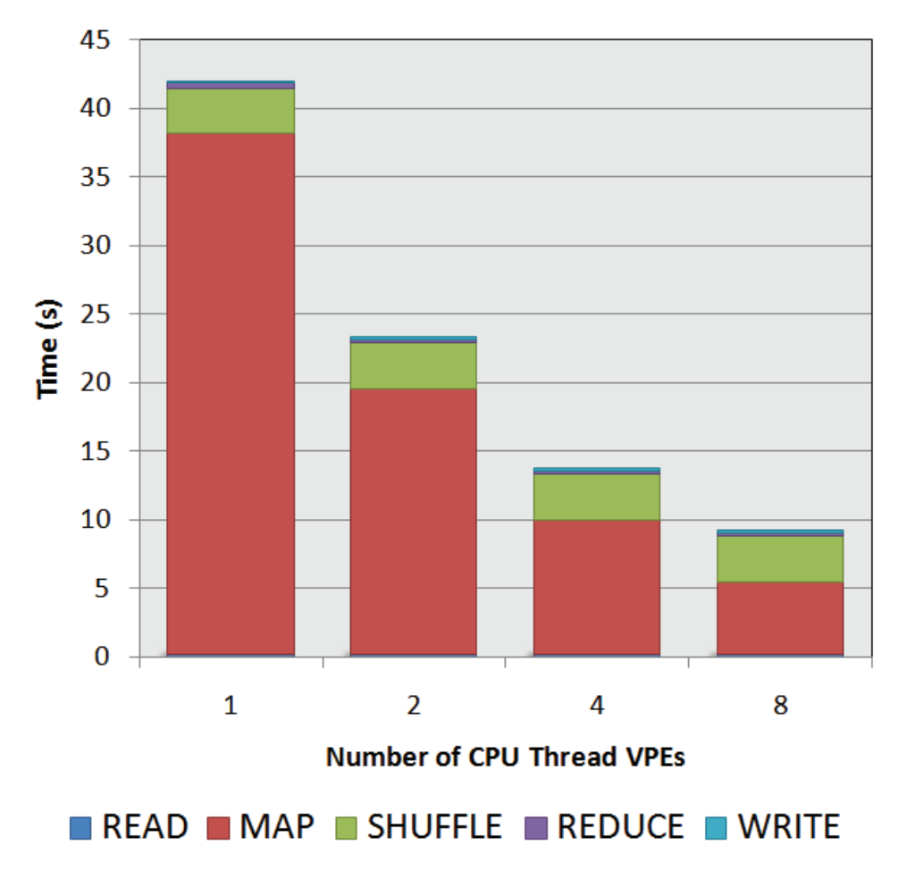
#CPU Cores	#GPU Cores	Time (s)
1	1	132
1	2	101
2	1	125
2	2	72
4	1	127
4	2	71
4	3	49
8	1	113
8	2	69

SHOW CASE 3 - MAP/REDUCE WORD COUNT

Map/Reduce in HyperFlow



Timing Results
Input: 83 million words



Two execution scenarios need to be addressed: (1) an upstream module, which executes sequentially, but generates output for concurrent execution downstream and (2) a downstream module which can only be executed sequentially and has to collect flows from upstream modules executed in parallel. Scenario (1) occurs when data are moved from the reader to mappers, as well as from the shuffling phase to reducers. This is necessary to enable data parallelism in the Map and Reduce phases. Scenario (2) happens when the execution comes back from the Map and Reduce to the Shuffling and Writer.
HyperFlow supports both scenarios: (1) an upstream can generate multiple flows with unique identifications to trigger data-parallelism in the pipeline; (2) TOM can limit task implementations to run sequentially by using a global lock. These are illustrated in the source code on the right.

```cpp
class IReader: public TaskImplementation {
  virtual int process(ExecutionContext *ctx,
                      TaskData *input, TaskData *output) {
    Data *data[N];
    // Read input file and place them into data[]
    ...
    // Then construct flows downstream
    for (int i=0; i<N; i++) {
      // We are not passing any reference flow so that
      // newly created flows will have unique id
      Flow *flow = ctx->createFlowForOutputPort(0);
      flow->data = data[i];
      output->addFlow(flow);
    }
    return 0;
  }
};

class MCombine: public TaskOrientedModule {
public:
  int          numberOfMappers;
  SafeCounter doneCount;
  Mutex        mutex;
  ListOfData   data;
};

class ICombine: public TaskImplementation {
  virtual int process(ExecutionContext *ctx,
                      TaskData *input, TaskData *output) {
    MCombiner *M = (MCombiner*)ctx->module;
    // Locking resource to accumulate
    M->mutex.lock();
    M->data.push_back(input->flows[0]->data);
    M->mutex.unlock();

    if (M->doneCounter->increase() < M->numberOfMappers)
      return -1;

    // Process data in module->data
    ...
    return 0;
  }
};
```