cudaFlow: A Modern C++ Programming Model for GPU Task Graph Parallelism

Tsung-Wei (TW) Huang and Dian-Lun (Luan) Lin
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Agenda

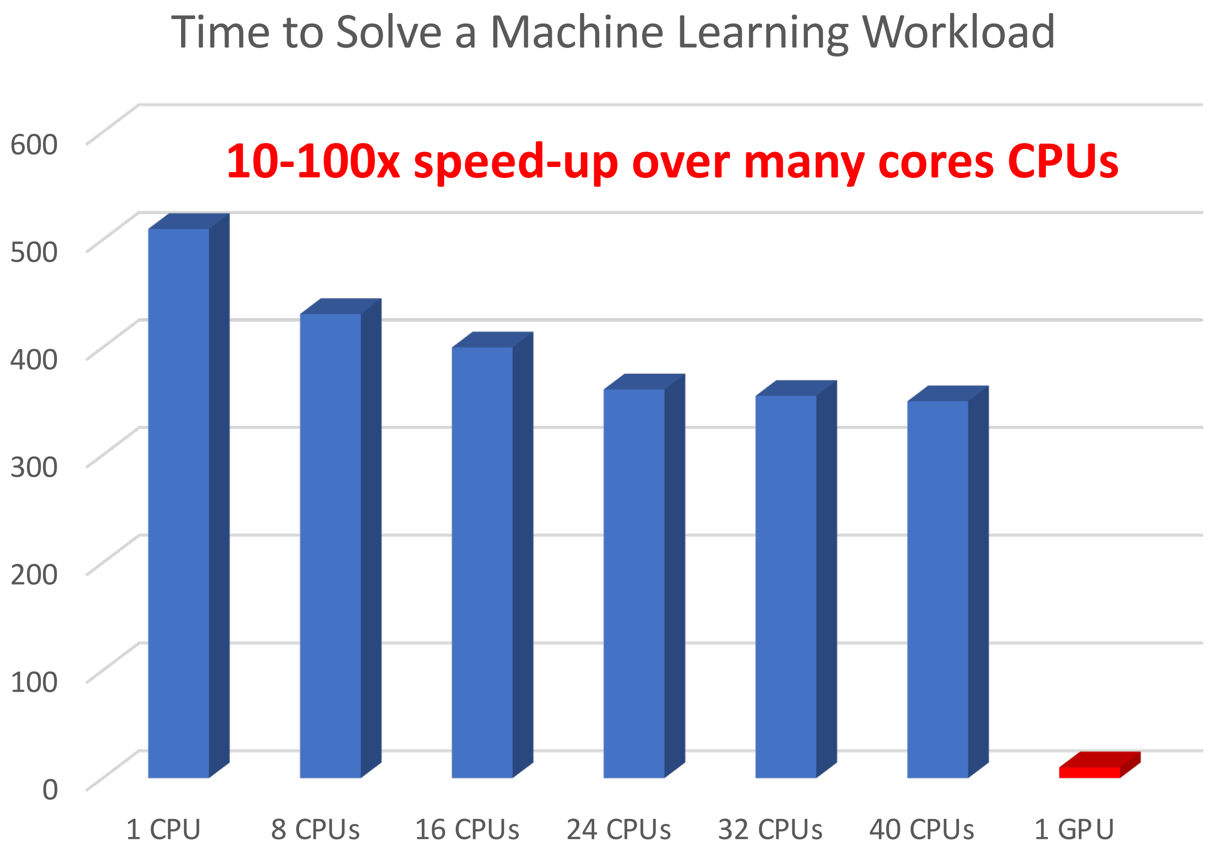
- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

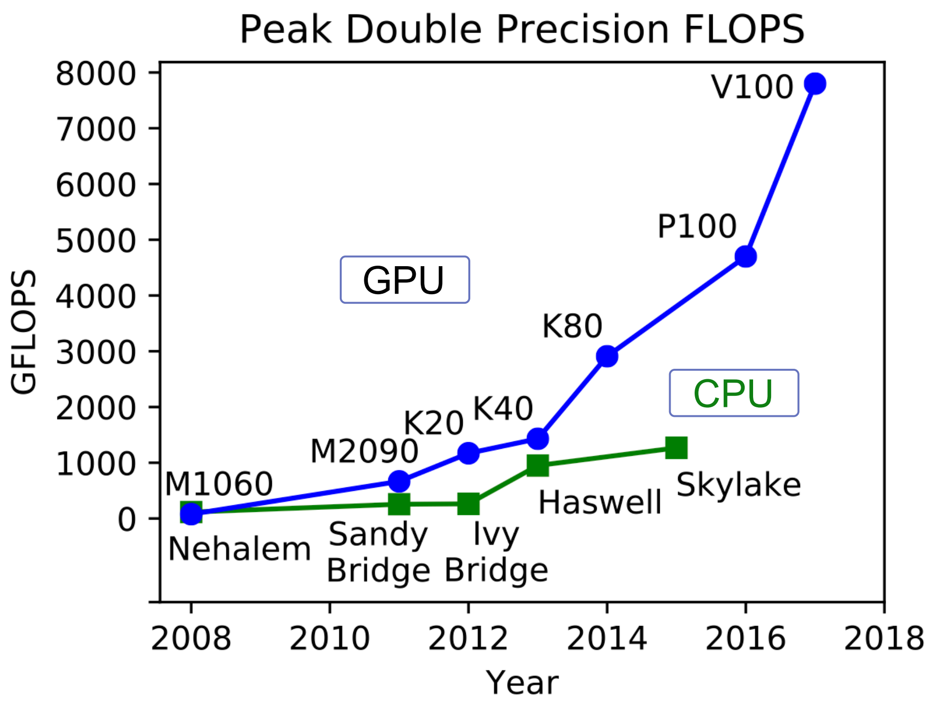
Agenda

- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

Why GPU Computing?

- GPU has advanced scientific computing to a new level

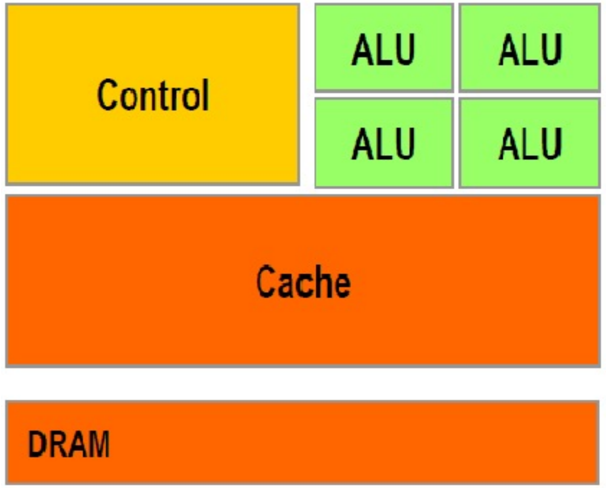




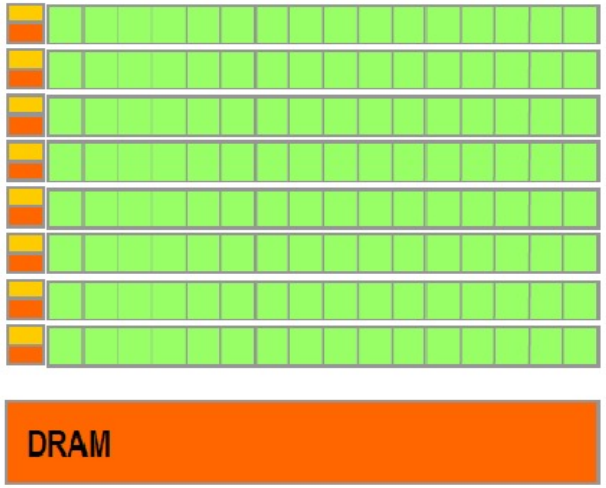
Over **60x** speedup in neural network training since 2013

CPU vs GPU

- CPU is built for compute-driven applications
  - A few *powerful* threads to compute critical control-flow blocks very fast
- GPU is built for throughput-driven applications
  - Many *lightweight* threads to compute large volume of data very fast



CPU

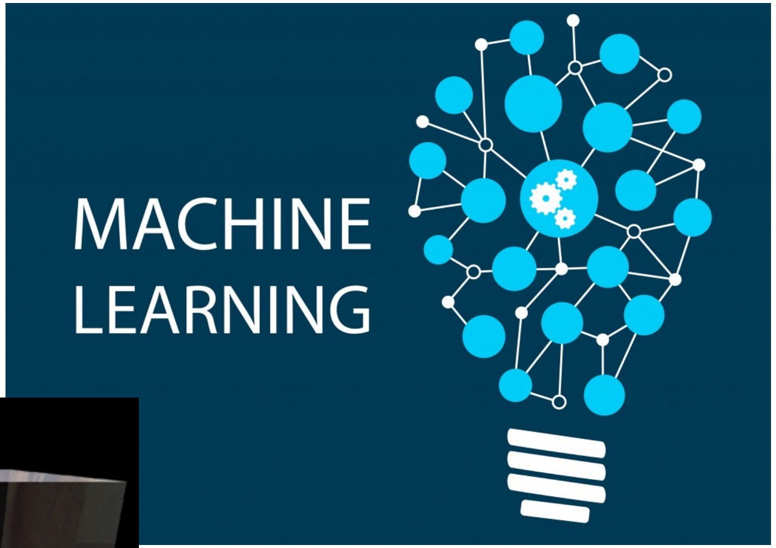


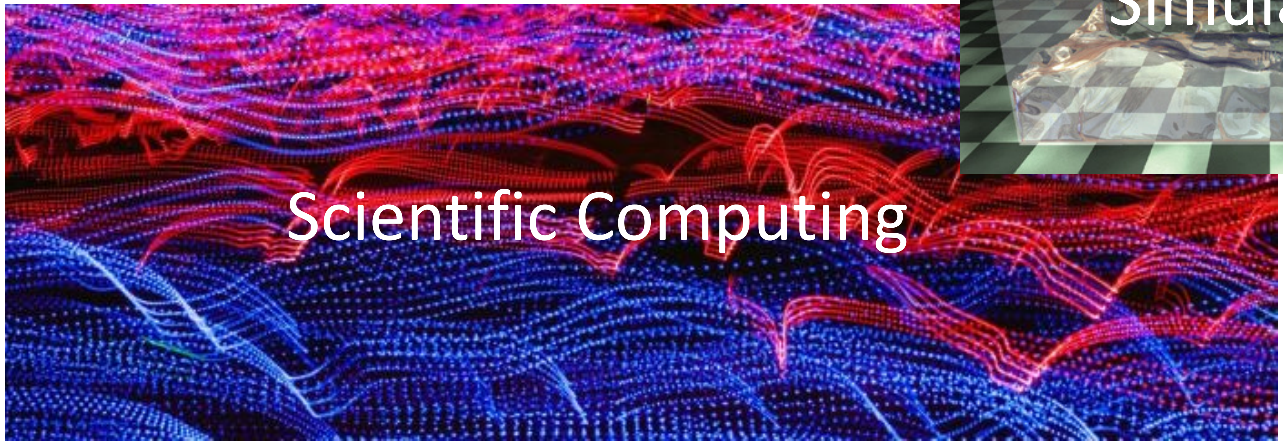
GPU

Nvidia RTX 6000
GPU card

Intel i7 CPU

GPU Application Landscape





Programming GPU

- Compute-unified device architecture (CUDA) model

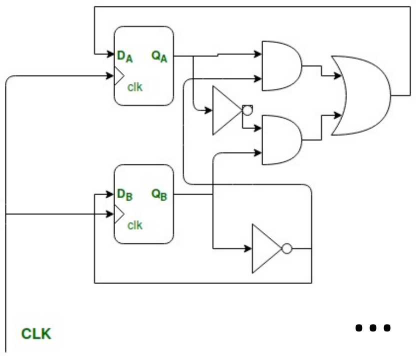
```
// saxpy.cu (single-precision A·X Plus Y)
__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) {
    y[i] = a*x[i] + y[i];
  }
}
// calling the saxpy kernel with grid, block, and shm
saxpy<<<grid, block, shm, stream>>>(n, a, x, y);
```

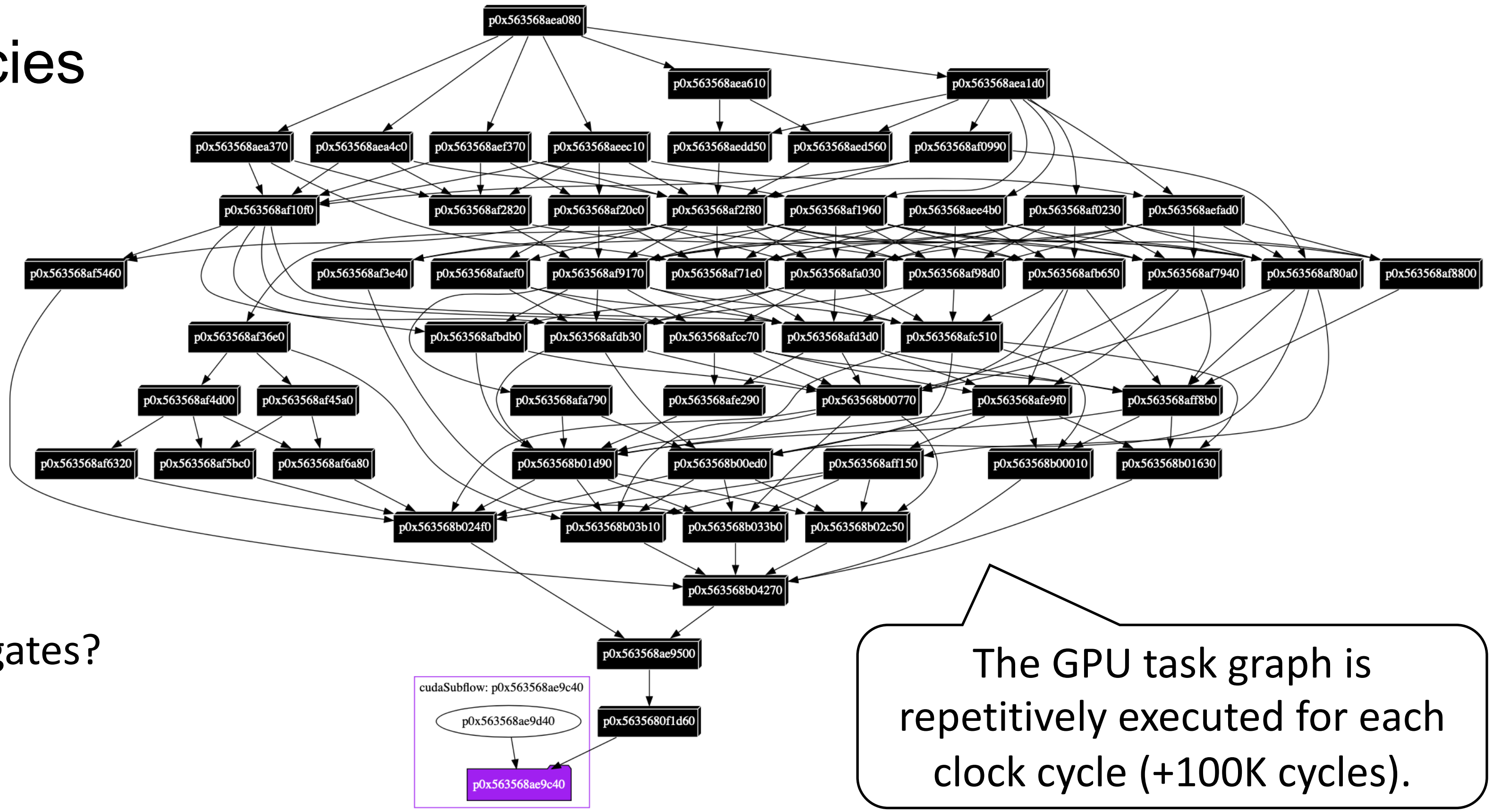
```
// use nvidia cuda compiler to compile the code
~$ nvcc saxpy.cu –o saxpy
```

a	2	2	2	2	2	2	2	2	2	2
*****										
x	2	3	2	1	3	2	3	2	1	2
+										
y	1	1	2	3	1	1	2	3	1	1
y	5	7	6	5	7	5	8	7	3	5

Today’s GPU Workload is Very Complex

- GPU-accelerated circuit simulation task graph
  - >100 kernels
  - >100 dependencies
  - >500s to finish



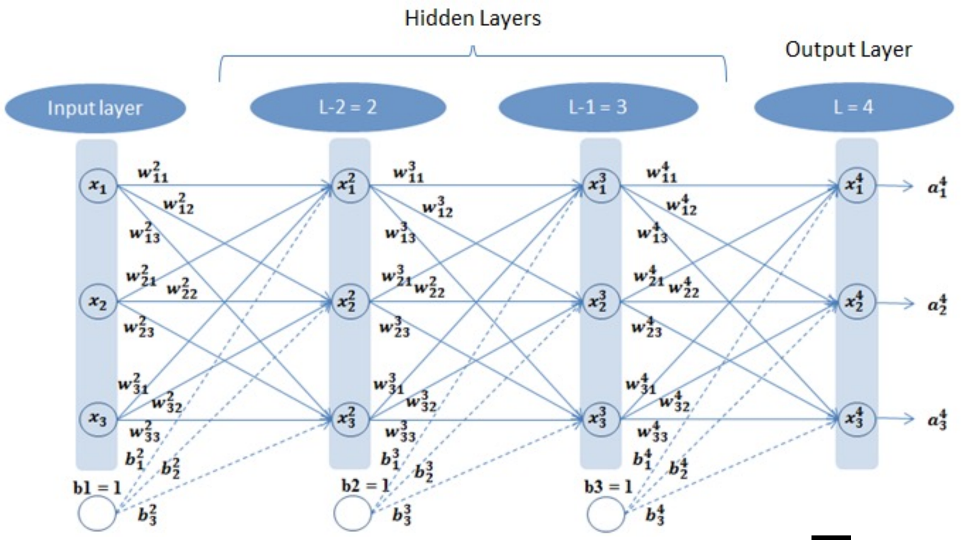


What are the output values of gates?
(+500M gates in nvdla designs https://github.com/nvdla)

The GPU task graph is repetitively executed for each clock cycle (+100K cycles).

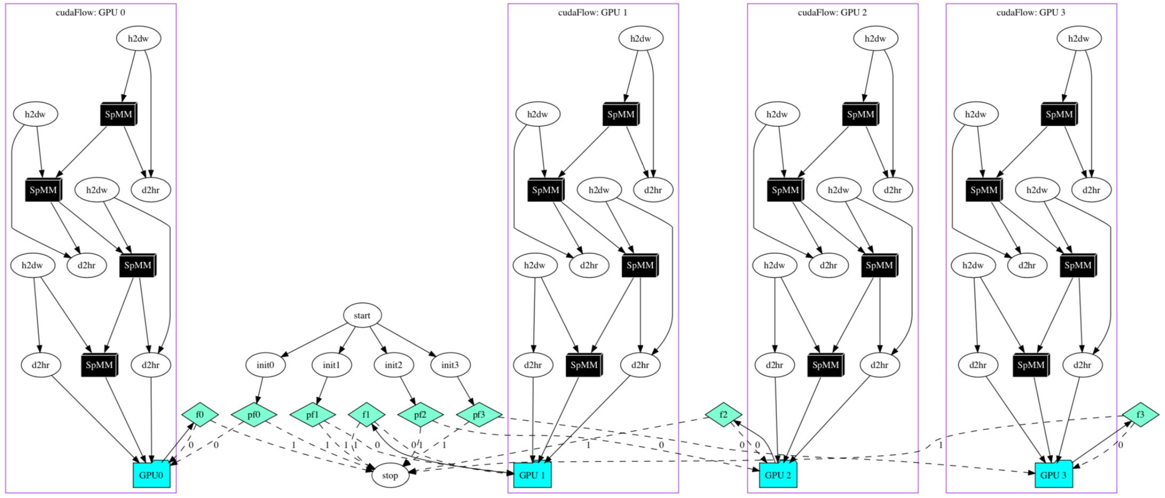
Another Example in Machine Learning

- Large neural network inference GPU task graph



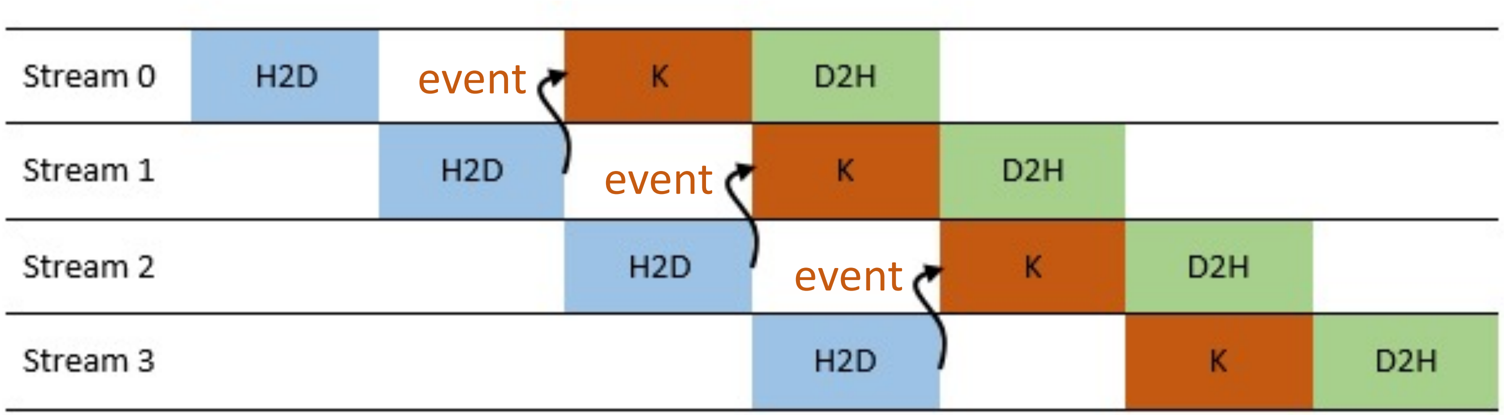
- Billions of parameters
- >1000 kernels
- >2000 dependencies
- Hours to finish

The GPU task graph is repetitively executed for millions (or infinite amount) of data batches.



CUDA Execution Model: Stream

- Launch a kernel through an asynchronous stream
  - Launch a kernel (e.g., my_kernel<<<grid, block, shm, stream>>>)
  - Run a kernel (e.g., **__global__** my_kernel())
- The “stream” variable keeps a sequence of kernel tasks to run
  - A stream is essentially an in-order queue (like std::queue)
  - A stream can synchronize with others through “events” (dependency)




```
// example stream APIs
cudaStreamCreate
cudaStreamMemcpyAsync
cudaStreamSynchronize
cudaEventRecord
…
```

Pros and Cons of Stream-based Execution

- Pros: Enable asynchronous execution to better utilize GPU
  - Memory copies overlap with kernel execution
  - Individual kernels running on different streams can overlap
- Cons: Incur *per-operation* overhead at each stream
  - The overhead can become significant for iterative GPU workloads

```
for(int step=0; step<1000000; step++){
  for(int krnl=0; krnl<1000000; krnl++){
    MyKernel<<<grid, block, shm, stream>>>(out_d, in_d);
  }
  cudaStreamSynchronize(stream);
}
```

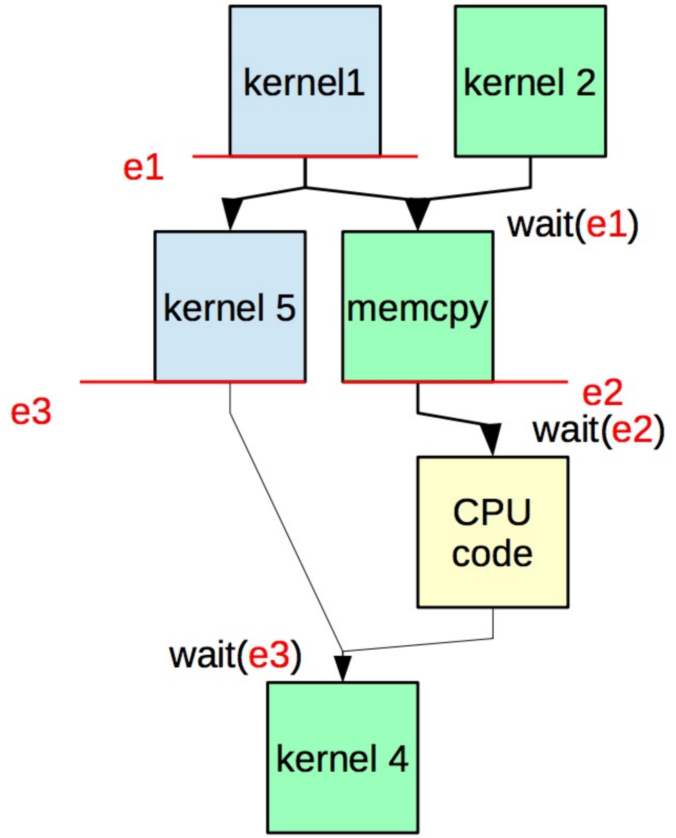
Execution overhead

Synchronization overhead

Task Dependency Graph is Hard to Build

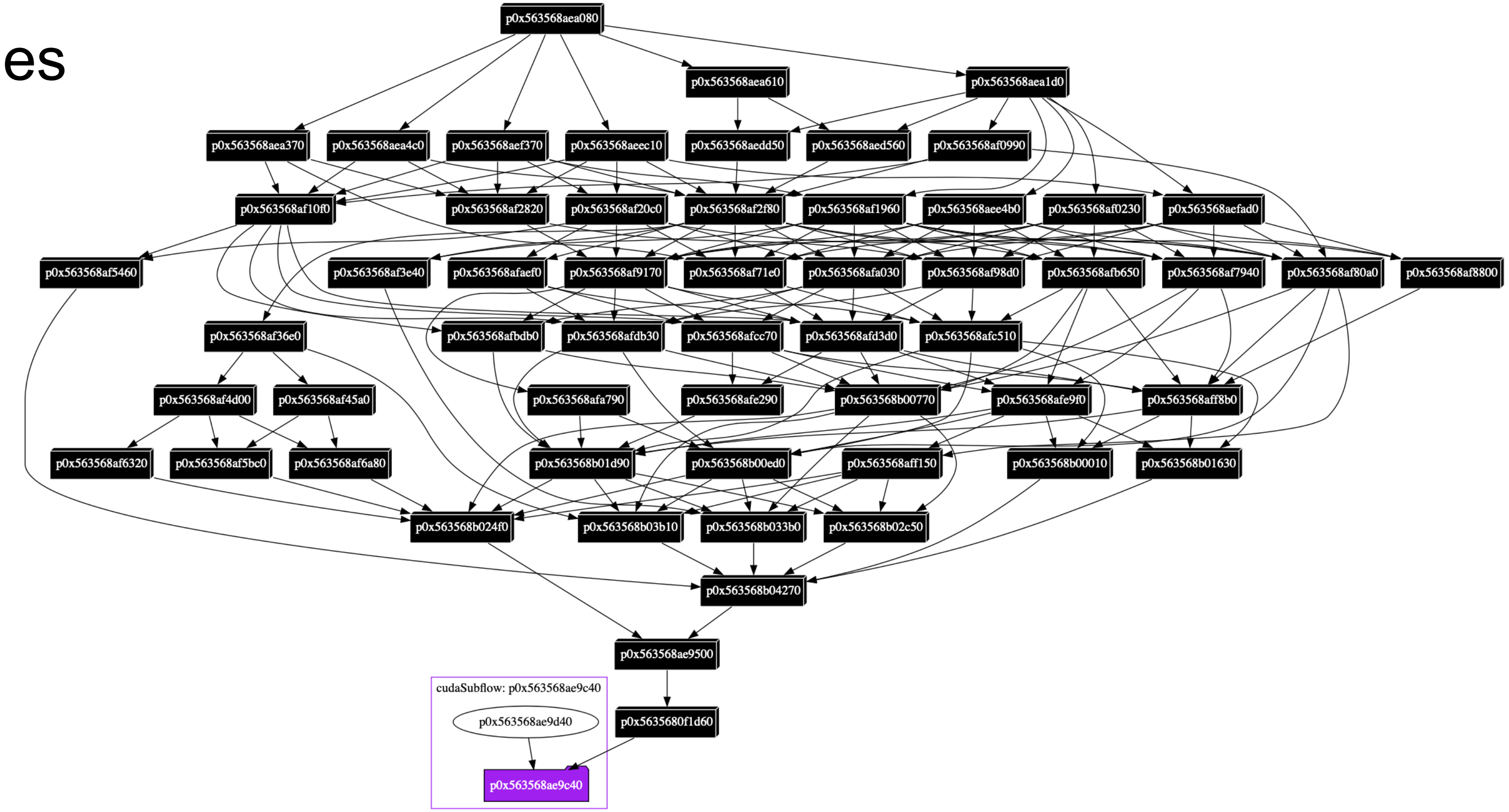
- Need to insert explicit events between GPU operations at different streams
- GPU runtime can't see tasks ahead to perform whole-graph optimization

```
// using streams to build a task dependency graph
kernel1<<>>();
cudaEventRecord(e1, a);
kernel2<<>>();
cudaStreamWaitEvent(b, e1);
cudaMemcpyAsync(,,,,b);
cudaEventRecord(e2, b);
kernel5<<>>();
cudaEventRecord(e3, a);
cudaEventSynchronize(e2);
// doing some CPU code to overlap kernel 5 via e2 and e3
cudaStreamWaitEvent(b, e3);
kernel4<<>>();
```



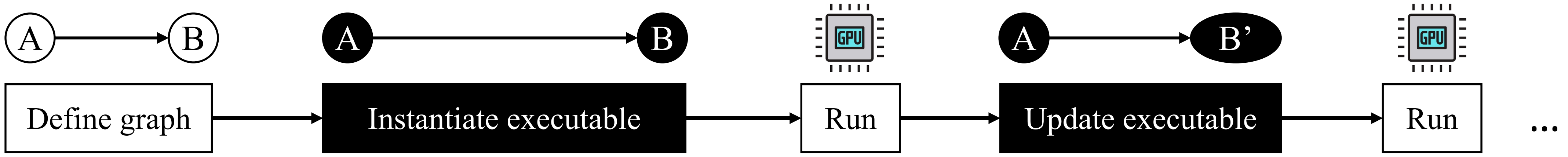
What About Large GPU Task Graphs?

- GPU-accelerated circuit simulation task graph
  - >100 kernels
  - >100 dependencies



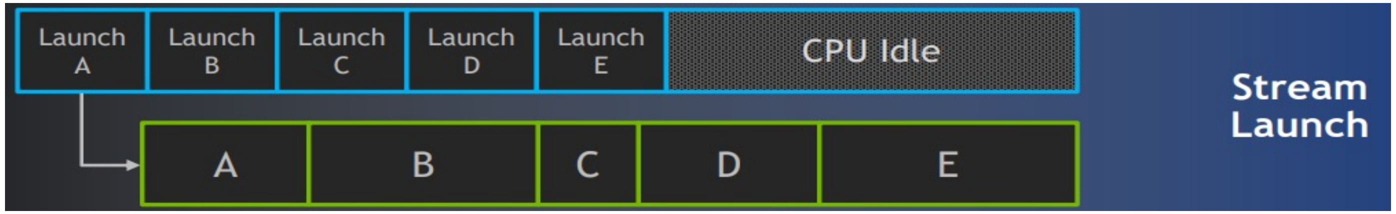
CUDA Execution Model: CUDA Graph

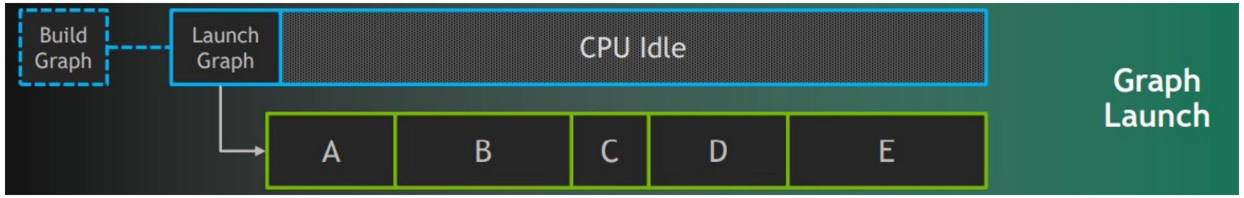
- Run a GPU workload using CUDA Graph with three steps
  1. Define an in-memory representation of the task dependency graph
     - Each node represents a GPU operation (e.g., memory copy, kernel)
     - Each edge represents a dependency
  2. Instantiate an optimized executable graph from a defined graph
  3. Launch the executable graph and update parameters between runs
     - Launch the executable graph requires only a single CPU call
     - CUDA runtime will perform automatic scheduling optimization

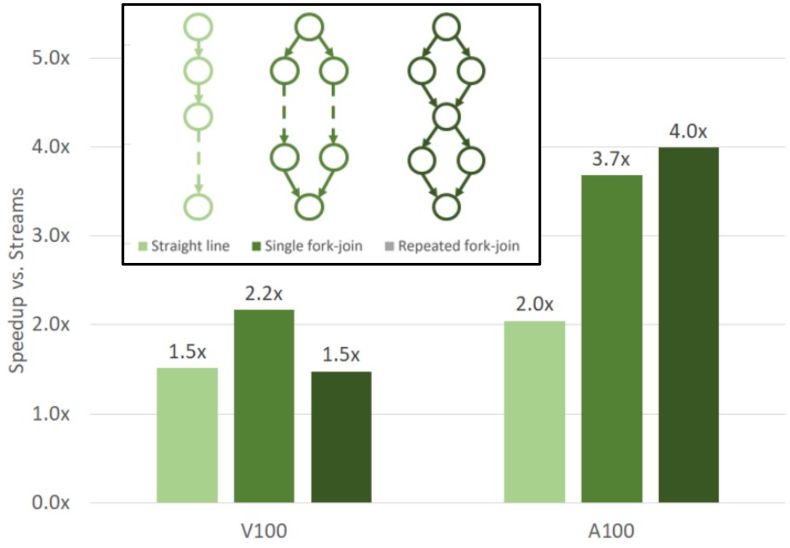


Comparison to Stream-based Execution

- CUDA Graph removes stream launch overhead for iterative patterns
  - Launch a CUDA graph requires only a single CPU call
  - CUDA runtime can perform the whole-graph optimization
  - New GPU architectures (e.g., A100) have many task graph optimizations



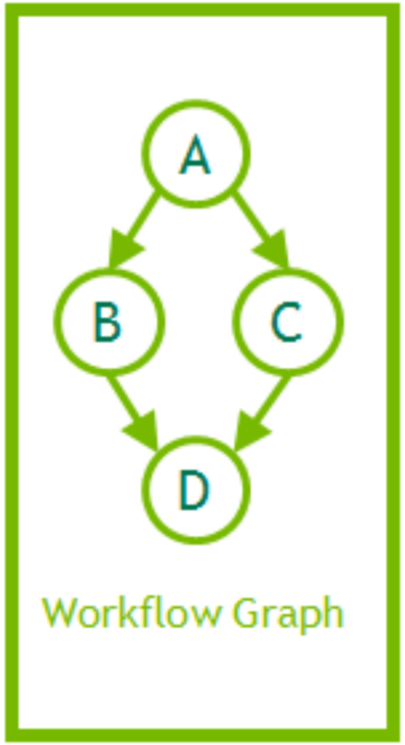




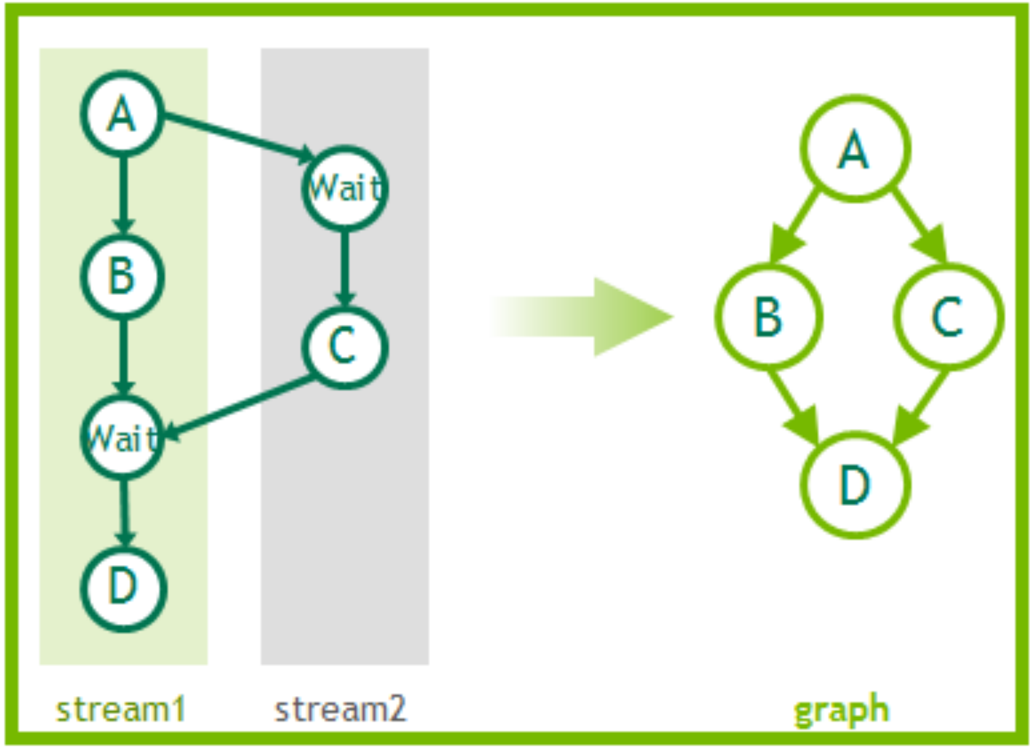
Ampere architecture white paper performance report: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Two Ways to Build a CUDA Graph

- Explicit CUDA Graph construction
- Implicit CUDA Graph construction



Explicit

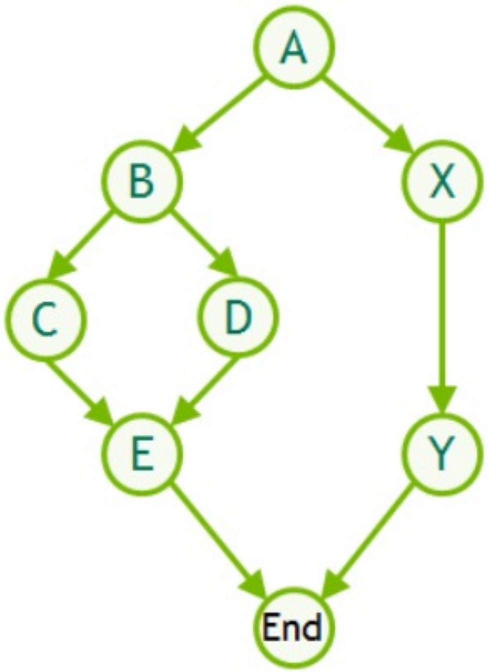


Implicit

Explicit CUDA Graph Construction

- Users define a CUDA graph explicitly using CUDA Graph API

```
// Graph data structure
cudaGraph_t                  // CUDA graph (opaque)
cudaGraphNode_t              // CUDA graph node (opaque)
cudaKernelNodeParams         // CUDA GPU kernel node parameters
…
// Explicit graph construction API
cudaGraphCreate              // Creates a graph
cudaGraphAddMemcpyNode       // Creates a memcpy node
cudaGraphAddKernelNode       // Creates a kernel execution node
cudaGraphGetNodes            // Returns a graph's nodes
cudaGraphInstantiate         // Creates an executable graph from a graph
cudaGraphLaunch              // Launches an executable graph in a stream
cudaGraphExecDestroy         // Destroys an executable graph
cudaGraphDestroy             // Destroys a graph
...
```

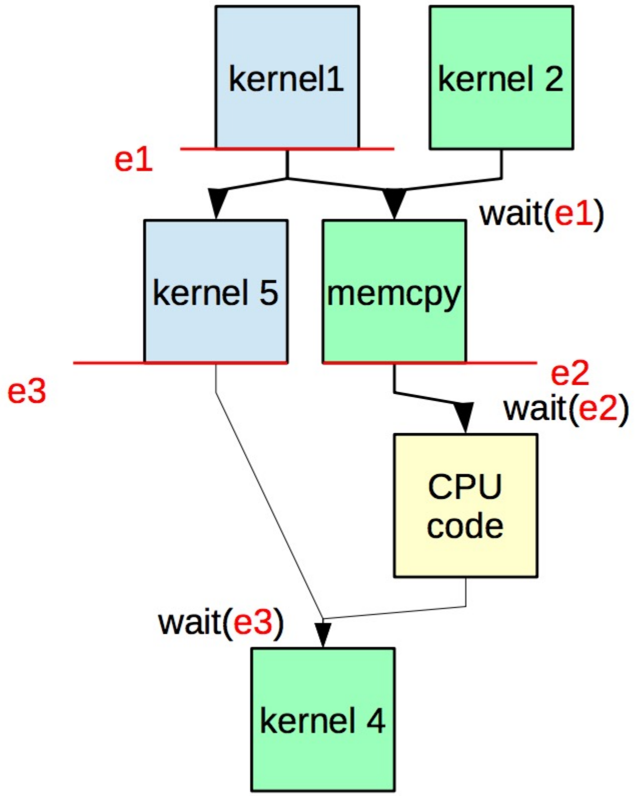


Implicit CUDA Graph Construction

- Users capture a CUDA graph implicitly through existing streams

```
cudaGraph_t graph;
cudaStreamBeginCapture(a);              // begin capturing a CUDA graph
kernel1<<…>>();
cudaEventRecord(e1, a);
kernel2<<…>>();
cudaStreamWaitEvent(b, e1);
cudaMemcpyAsync(,,,,b);
kernel5<<…>>();
cudaEventRecord(e3, a);
cudaLaunchHostFunc(b, cpucode, params);
cudaStreamWaitEvent(b, e3);
kernel4<<…>>();
cudaStreamEndCapture(a, &graph);        // end capturing a CUDA graph
cudaGraphInstantiate(…);
…
```

use stream to execute dependent GPU operations as before



Comparison between Explicit and Implicit Methods

- Explicit CUDA Graph construction
  - ☺ straightforward graph definition identical to an application task graph
  - ☺ performance is typically the best
  - ☹ extremely tedious to program
    - Flat parameter structure and CUDA Graph API produce a lot of boilerplate code
    - Often result in 2-10x increase of the codebase
  - ☹ can only handle GPU workloads with known parameters
- Implicit CUDA Graph capturing
  - ☺ flexible in getting a CUDA graph from existing stream-based code
  - ☹ if that code doesn't exist, you need to manage streams and events

    !!! CUDA Graph performance is highly dependent on the stream assignment
  - ☹ not easy to adapt code to new application task graphs

cudaFlow Project Mantra

How can we streamline the programming of CUDA Graph while encapsulating technical details between an application task graph and its native CUDA graph?

Agenda

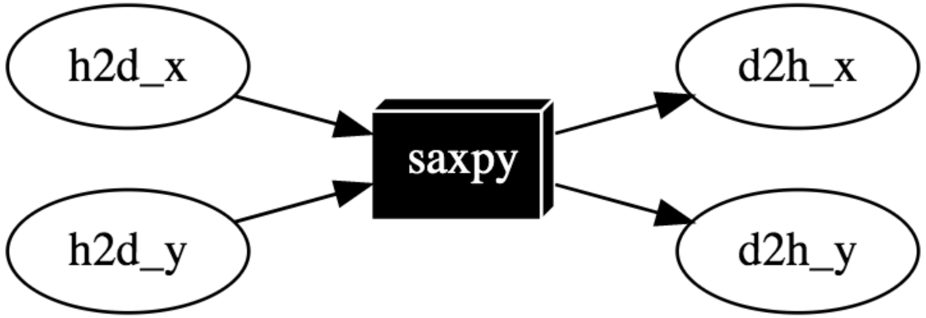
- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

An Explicit Saxpy Task Graph in cudaFlow

```
// saxpy (single-precision A·X Plus Y) kernel
__global__ void saxpy(int n, float a, float *x, float *y) {
  if (int i = blockIdx.x*blockDim.x + threadIdx.x; i < n) {
    y[i] = a*x[i] + y[i];
  }
}
```

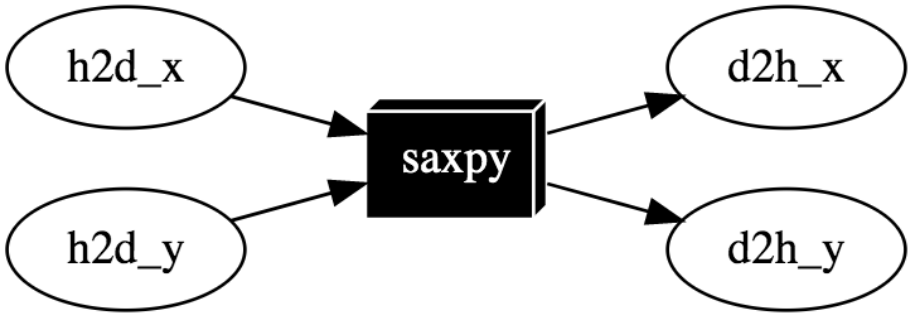
cudaFlow maintains an 1-to-1 mapping between the application task graph and a native CUDA graph

```
// create an explicit saxpy task graph using cudaFlow
tf::cudaFlow cf;
tf::cudaTask h2d_x = cf.copy(dx, hx, N);
tf::cudaTask h2d_y = cf.copy(dy, hy, N);
tf::cudaTask d2h_x = cf.copy(hx, dx, N);
tf::cudaTask d2h_y = cf.copy(hy, dy, N);
tf::cudaTask saxpy  = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
cf.offload();
```



An Implicit Saxpy Task Graph in cudaFlow

```
// capture an implicit saxpy task graph using "cudaFlowCapturer"
tf::cudaFlowCapturer cf;
tf::cudaTask h2d_x = cf.copy(dx, hx, N);
tf::cudaTask h2d_y = cf.copy(dy, hy, N);
tf::cudaTask d2h_x = cf.copy(hx, dx, N);
tf::cudaTask d2h_y = cf.copy(hy, dy, N);
tf::cudaTask saxpy  = cf.on([&](cudaStream stream){
  // you can capture the saxpy kernel if you know all the kernel execution parameters (e.g., grid)
  saxpy<<<(N+255)/256, 256, 0, stream>>>(N, 2.0f, dx, dy)
  // or you can capture the saxpy kernel through a public stream-based API
  saxpy_through_a_stream_based_API(N, 2.0f, dx, dy)
});
kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
cf.offload();
```

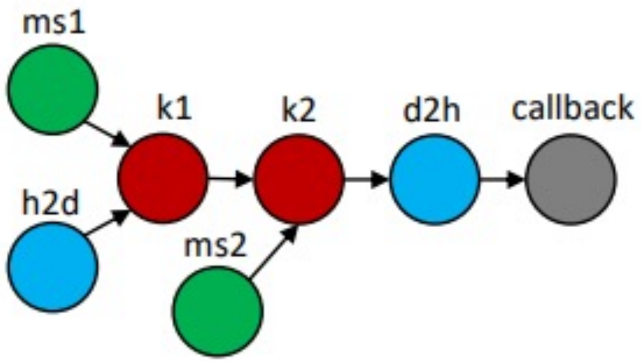


cudaFlowCapturer automatically performs optimization (e.g., deciding tedious stream and event insertions) to transform the application task graph to a native CUDA graph.

Why cudaFlow?

- A slightly more complicated task graph can blow up your CUDA Graph code

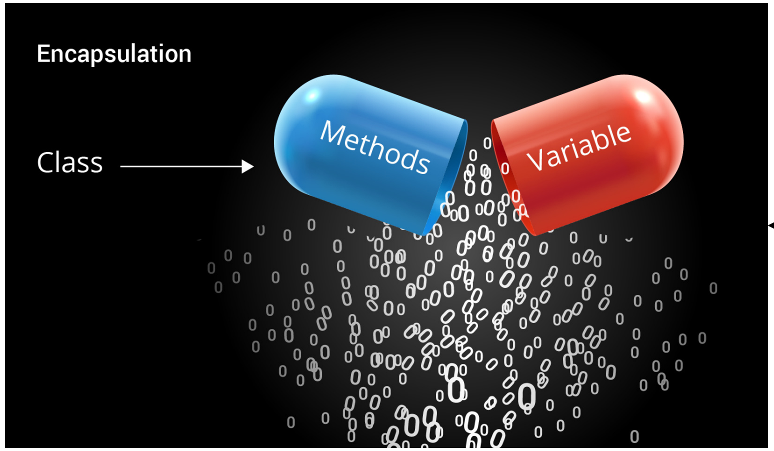
```
cudaStream_t streamForGraph;
cudaGraph_t graph;
std::vector<cudaGraphNode_t> nodeDependencies;
cudaGraphNode_t memcpyNode, kernelNode, memsetNode;
checkCudaErrors(cudaStreamCreate(&streamForGraph));
cudaKernelNodeParams kernelNodeParams = {0};
cudaMemcpy3DParms memcpyParams = {0};
cudaMemsetParams memsetParams = {0};
memcpyParams.srcArray = NULL;
memcpyParams.srcPos = make_cudaPos(0, 0, 0);
memcpyParams.srcPtr =
    make_cudaPitchedPtr(inputVec_h, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.dstArray = NULL;
memcpyParams.dstPos = make_cudaPos(0, 0, 0);
memcpyParams.dstPtr =
    make_cudaPitchedPtr(inputVec_d, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.extent = make_cudaExtent(sizeof(float) * inputSize, 1, 1);
memcpyParams.kind = cudaMemcpyHostToDevice;
checkCudaErrors(cudaGraphCreate(&graph, 0));
checkCudaErrors(
  cudaGraphAddMemcpyNode(&memcpyNode, graph, NULL, 0, &memcpyParams
));
//... more than 100 lines of code to follow
```




```
cudaFlow cf;
cudaTask h2d = cf.copy(inputVec_d, inputVec_h, inputSize);
cudaTask ms1 = cf.memset(outputVec_d, 0, input_size);
cudaTask ms2 = cf.memset(result_d, 0, 1);
cudaTask k1 = cf.kernel(reduce, inputVec_d, outputVec_d, inputSize);
cudaTask k2 = cf.kernel(reduce_final, outputVec_d, result_d);
cudaTask d2h = cf.copy(result_h, result_d, 1);
cudaTask callback = cf.host(fn, &hostFnData);
k1.succeed(h2d, ms1);
k2.succeed(k1, ms2);
k2.precede(d2h);
d2h.precede(callback);
```

cudaFlow Design Philosophy

- What cudaFlow and cudaFlowCapturer do
  - Encapsulate tasking details of dependent GPU operations
  - Build a GPU task graph (tasks, dependencies, updates)
  - Manage offload details (graph optimization, instantiation)
  - Clean up graph runtime storage
- What cudaFlow and cudaFlowCapturer *don't* do
  - Simply kernel programming
  - Abstract memory and data management
  - Develop yet another runtime



C++ Library developers should think carefully about what abstraction is mostly suitable for application developers

cudaFlow API Category

- Graph construction
  - Create a task graph of GPU operations
- Graph execution
  - Transform the application task graph to a native CUDA graph
  - Instantiate the executable graph
- Graph update
  - Update task parameters between successive offloads

cudaFlow API: Graph Construction

```
// create a kernel task
tf::cudaTask kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
```

```
// capture a kernel task through an internal stream
tf::cudaTask saxpy  = cf.on([&](cudaStream stream){ cuBLAS_API(stream, …); });
```

```
// create a memory set task
tf::cudaTask memset_target = cf.memset(target, 0, sizeof(int) * count);
tf::cudaTask same_as_above = cf.fill(target, 0, count);
```

```
// create a memory copy task
tf::cudaTask memcpy_target = cf.memcpy(target, source, sizeof(int) * count);
tf::cudaTask same_as_above = cf.copy(target, source, count);
```

```
// create a dependency between two tasks
memset_target.precede(kernel);
```

cudaFlow API: Graph Execution

```
// offload a cudaFlow
cf.offload();                                                   // run the cudaFlow once
cf.offload_n(10);                                               // run the cudaFlow 10 times
cf.offload_until([loops=5] () mutable { return loops-- == 0; }); // five times
```

```
// offload a cudaFlow capturer (additional transformation to a native CUDA graph is required)*
// define a transformation algorithms (round-robin with four streams)
cf.make_optimizer<tf::cudaFlowRoundRobinCapturing>(4);
cf.offload();                                                   // run the cudaFlow once
cf.offload_n(10);                                               // run the cudaFlow 10 times
cf.offload_until([loops=5] () mutable { return loops-- == 0; }); // five times
```

* Dian-Lun Lin and Tsung-Wei Huang, “Efficient GPU Computation using Task Graph Parallelism,” *European Conference on Parallel and Distributed Computing (Euro-Par)*, Portugal, 2021

cudaFlow API: Graph Update

```
// define a task dependency graph
tf::cudaTask task = cf.kernel(grid1, block1, shm1, my_kernel, args1...);
…
// offload the cudaFlow
cf.offload();
// update the parameter of a task previously created by the cudaFlow
cf.kernel(task, grid2, block2, shm2, my_kernel, args2...);
// offload the cudaFlow again with the same graph topology but new kernel parameters
cf.offload();
…
```

Each graph construction method comes with an overload to update parameters of a task previously created from the same method.

cudaFlow API: Graph Update (cont'd)

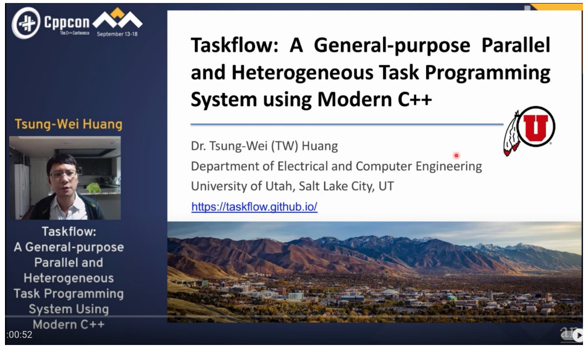
- Graph topology
  - Cannot change the graph topology of an offloaded cudaFlow
- Kernel task
  - Cannot change the kernel function but only its parameters
    - If a kernel is templated on an operator, use functor instead of lambda
  - Cannot change the kernel execution context
- Memory operation task
  - Cannot change the CUDA devices to which the operands came from
  - Cannot change the CUDA devices of source/target memory pointers

More details can be found at the page of CUDA Graph Runtime API: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

Integration to Taskflow

- cudaFlow can be used as a “cudaFlow task” in Taskflow*

```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N);});
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N);});

auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
  auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
  auto h2d_y = cf.copy(dy, hy.data(), N);
  auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
  auto d2h_y = cf.copy(hy.data(), dy, N);
  auto kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
  kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});

cudaflow.succeed(allocate_x, allocate_y);
executor.run(taskflow).wait();
```



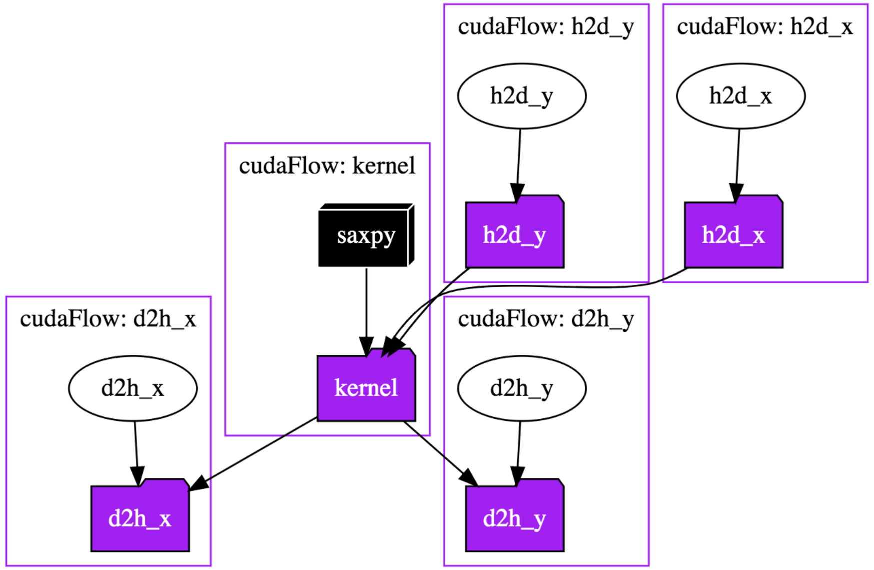
Taskflow@CppCon20: https://www.youtube.com/watch?v=MX15huP5DsM

* Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin, “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,” *IEEE TPDS*, 2021 [https://taskflow.github.io/]

Granularity Matters

```
tf::Task h2d_x = taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.copy(dx, hx.data(), N);
});
tf::Task h2d_y = taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.copy(dy, hy.data(), N);
});
tf::Task d2h_x = taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.copy(hx.data(), dx, N);
});
tf::Task d2h_y = taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.copy(hy.data(), dy, N);
});
tf::Task kernel = taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
});
kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
```

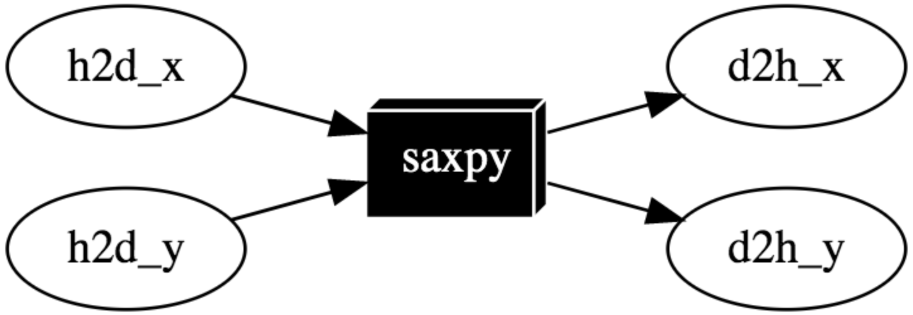
// Five cudaFlows to describe saxpy task graph



The static cost of CUDA Graph is non-negligible, typically at hundreds of million-seconds scale.

Granularity Matters (cont'd)

```
// one cudaFlow to describe the saxpy task graph
tf::cudaFlow cf;
tf::cudaTask h2d_x = cf.copy(dx, hx, N);
tf::cudaTask h2d_y = cf.copy(dy, hy, N);
tf::cudaTask d2h_x = cf.copy(hx, dx, N);
tf::cudaTask d2h_y = cf.copy(hy, dy, N);
tf::cudaTask saxpy  = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
cf.offload();
```



Putting together as many GPU operations in a CUDA graph as possible typically gives a better performance.

Place a cudaFlow on a Specific GPU

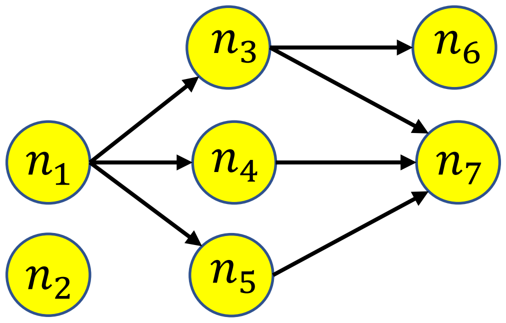
```
// create a cudaFlow is created under the default GPU context (GPU 0)
tf::cudaFlow cf_on_gpu0;
tf::cudaTask task = cf_on_gpu0.kernel(grid1, block1, shm1, my_kernel_1, args1...);

// create a cudaFlow under the context of GPU 2 using RAII-styled context switch
{
   tf::cudaScopedDevice gpu2(2);
   tf::cudaFlow gpu2;
   tf::cudaTask task = gpu2.kernel(grid2, block2, shm2, my_kernel_2, args2...);
}

// emplace a cudaFlow task under the context of GPU 3 using taskflow
taskflow.emplace_on([](tf::cudaFlow& cf){
   cf.kernel(…);
}, 3);
```

Agenda

- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

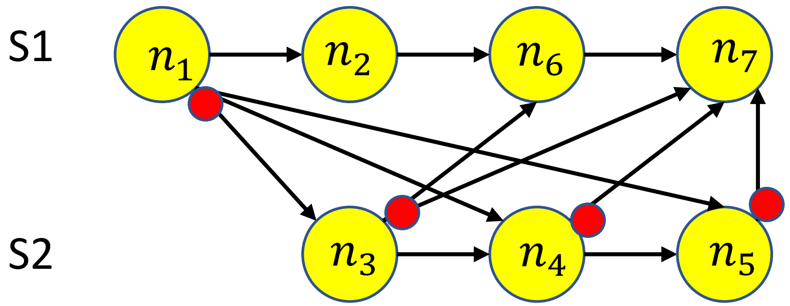
cudaFlow vs cudaFlowCapturer Execution

- cudaFlow is essentially a C++ wrapper over CUDA Graph
  - Always has an 1-to-1 mapping between cudaFlow and its CUDA graph
- cudaFlowCapturer instead captures the CUDA graph later
  - No guarantee to have 1-to-1 mapping due to closed kernel source code
    - cuBLAS, cuSparse, cuDNN, third-party kernel implementations, etc.
  - Need transformation from cudaFlowCapturer to a CUDA graph



Application graph
(cudaFlowCapturer)

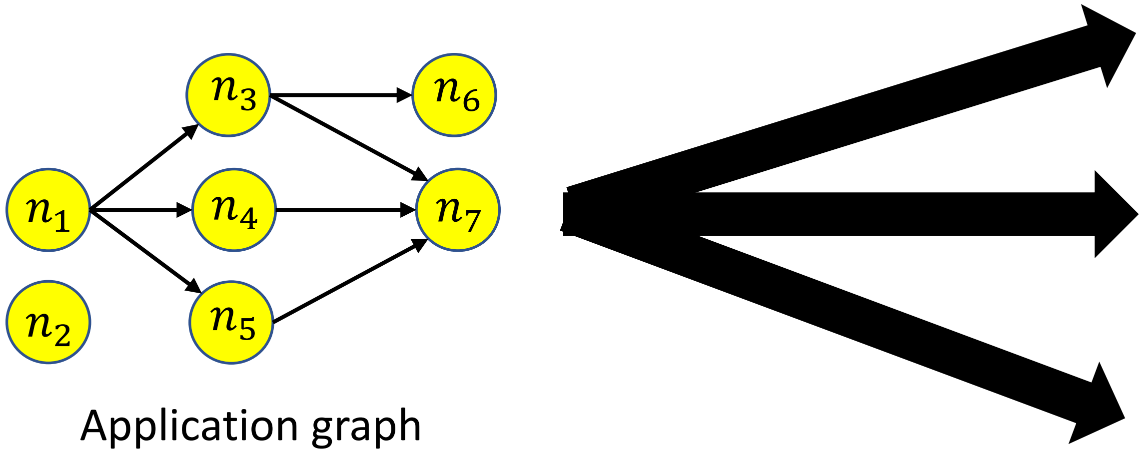
Transformation algorithm by the cudaFlow library



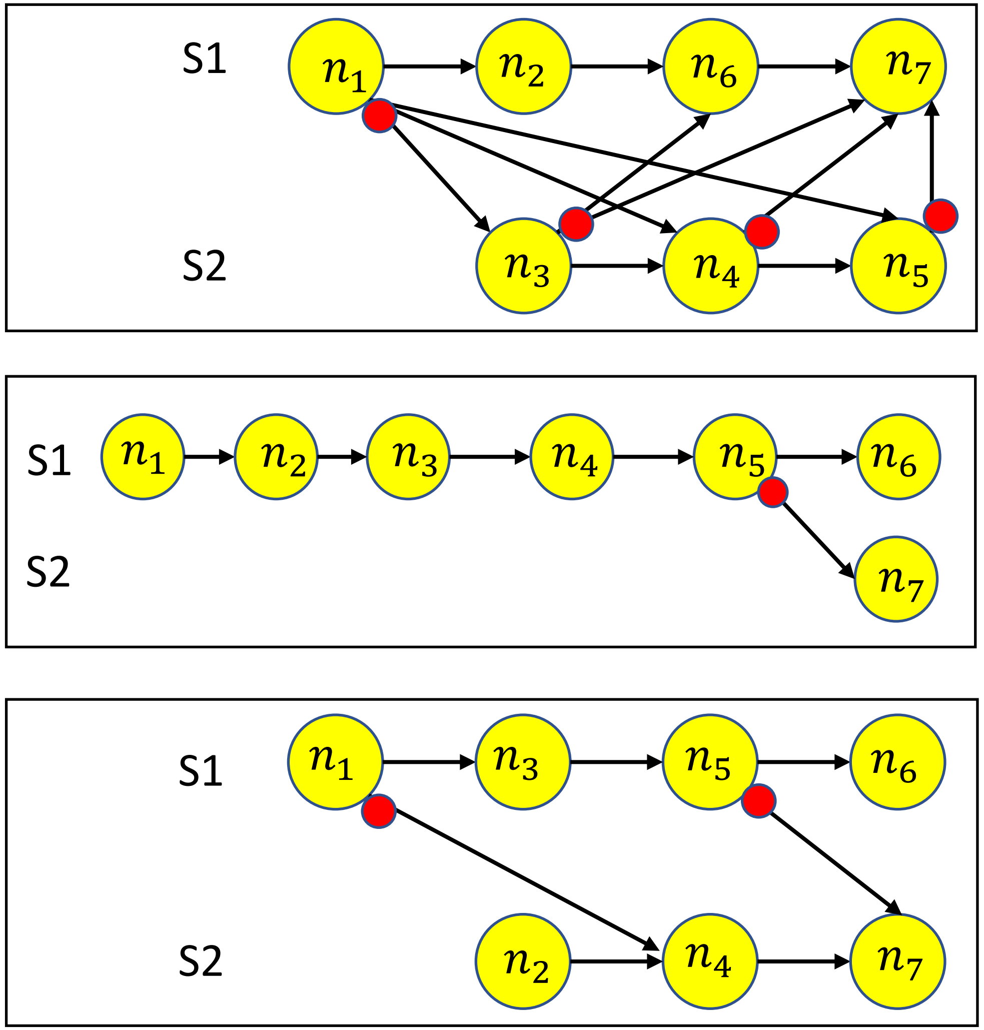
Transformed CUDA graph
(two streams and four events)

Objective of cudaFlowCapturer Transformation

- Multiple transformed graphs exist
- Objective of transformation
  - Achieve good load balancing
  - Minimize the transformed graph size



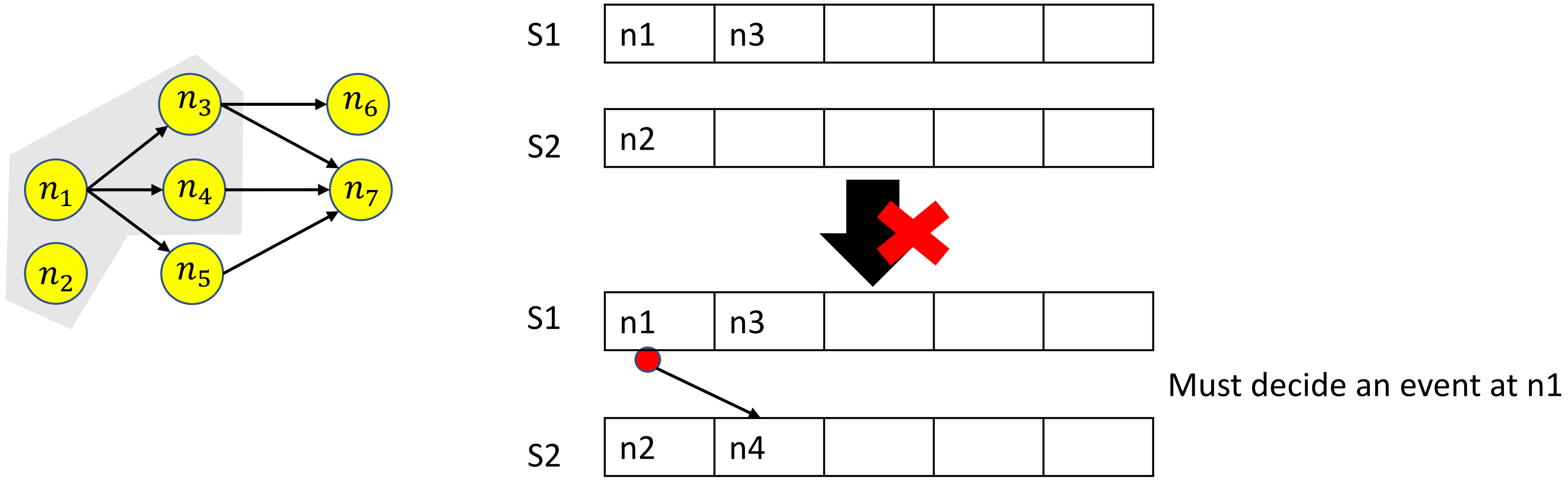
Application graph
(cudaFlowCapturer)



Each red point represents an CUDA event

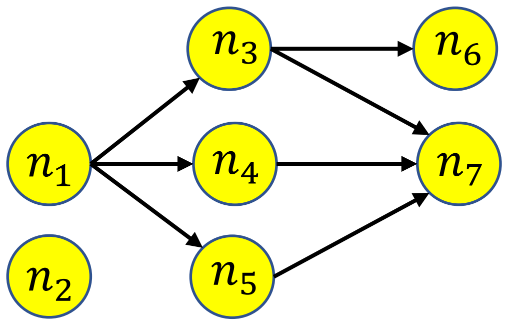
Key Challenge of Graph Transformation

- Streams are asynchronous and stateful
- Events can only be created by the last enqueued node
  - Dependency can only be created in a *forward* manner

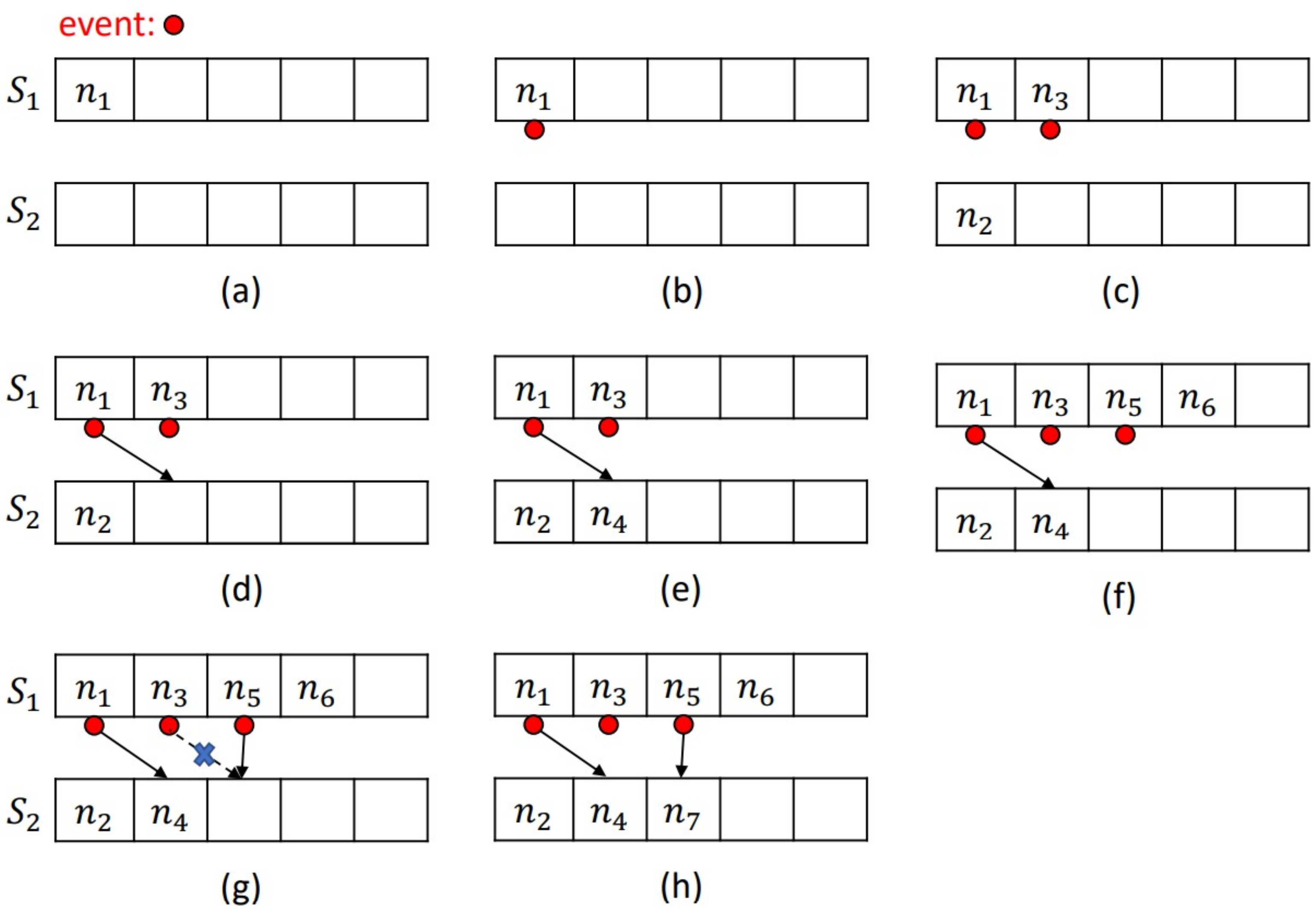


Graph Transformation Algorithm

1. Perform levelization
2. Loop from the lowest to the highest level, schedule nodes in round-robin (RR)
3. Create events based on the scheduled results

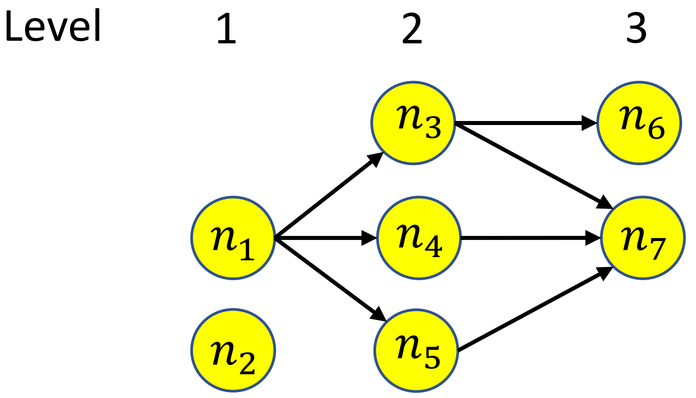


Round-robin stream assignment enables load balancing and look-ahead event creation

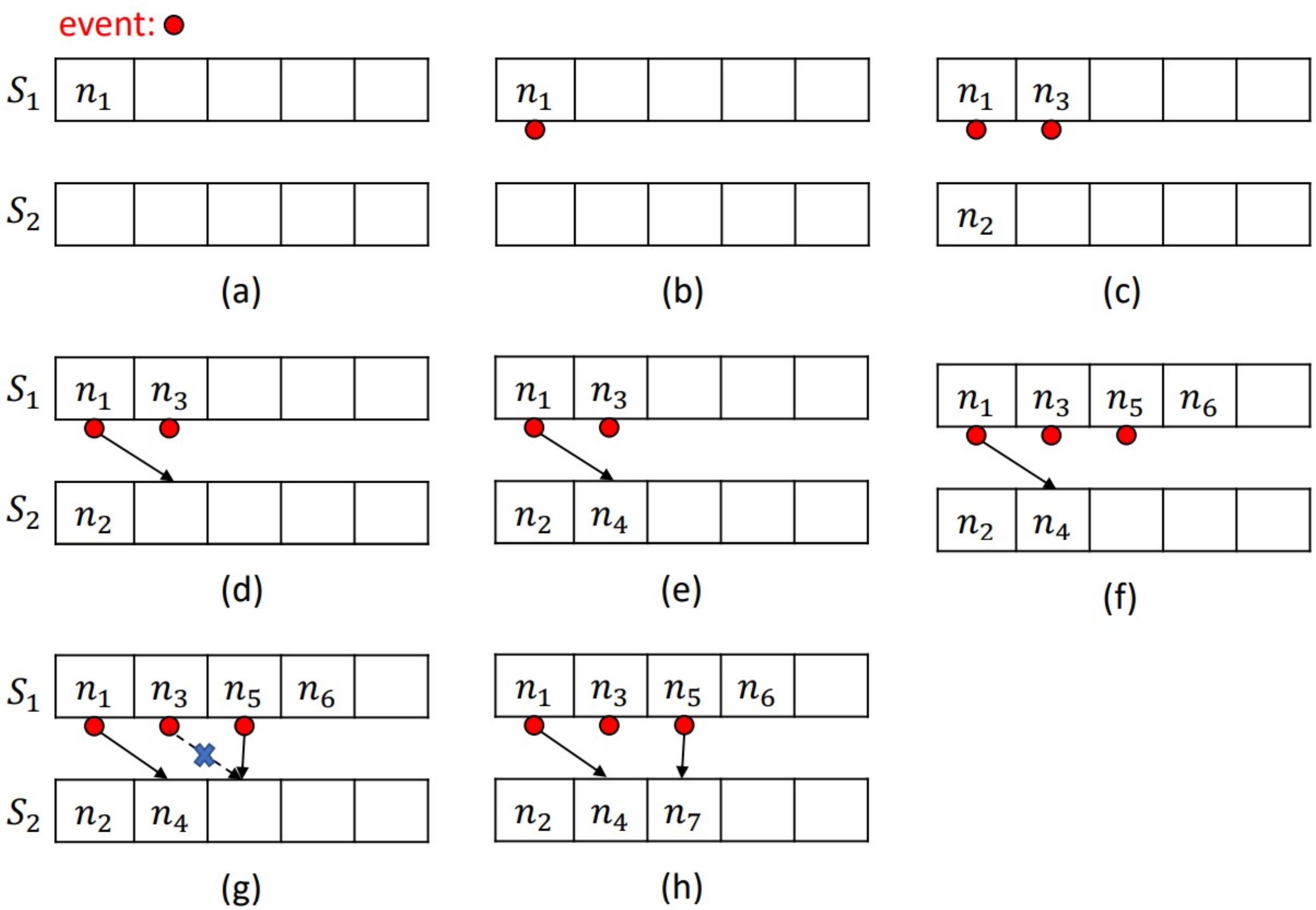


Graph Transformation Algorithm (cont'd)

1. Perform levelization
2. Loop from the lowest to the highest level, schedule nodes in round-robin (RR)
3. Create events based on the scheduled results

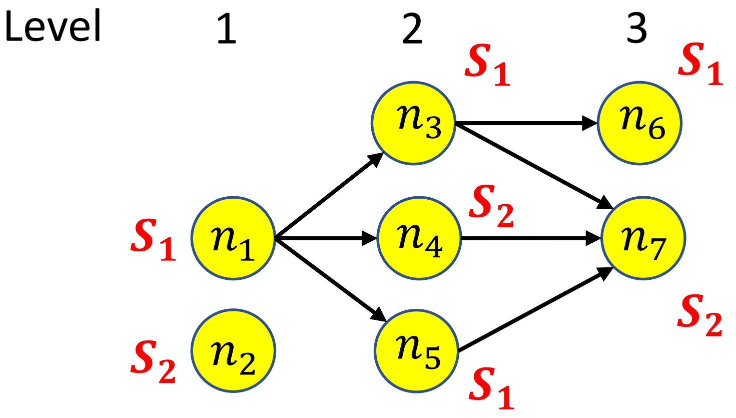


Round-robin stream assignment enables load balancing and look-ahead event creation

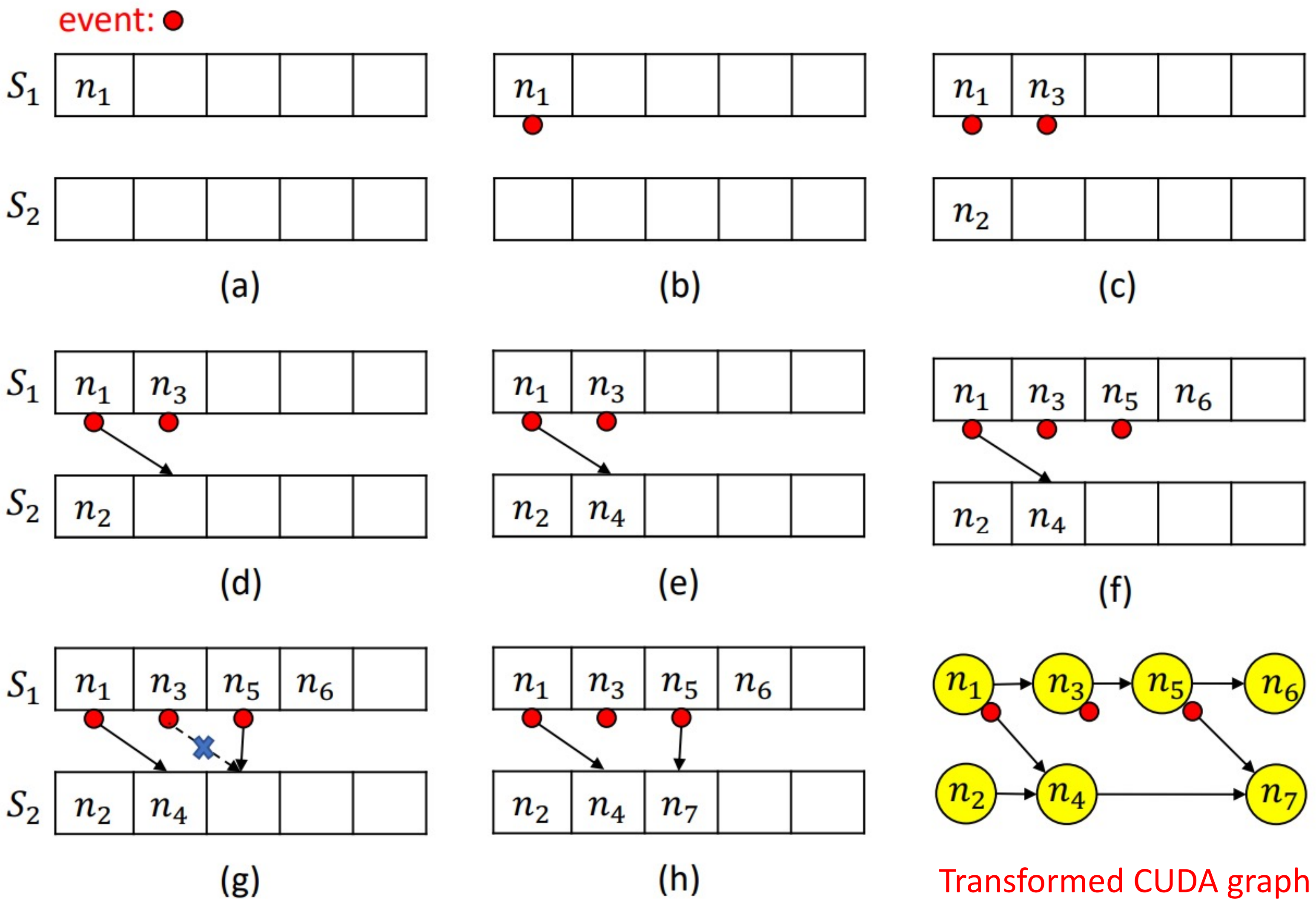


Graph Transformation Algorithm (cont'd)

1. Perform levelization
2. Loop from the lowest to the highest level, schedule nodes in round-robin (RR)
3. Create events based on the scheduled results



Round-robin stream assignment enables load balancing and look-ahead event creation



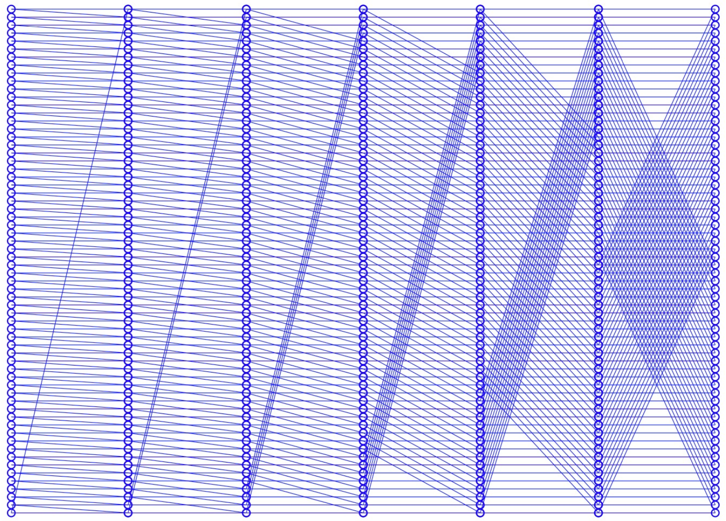
Agenda

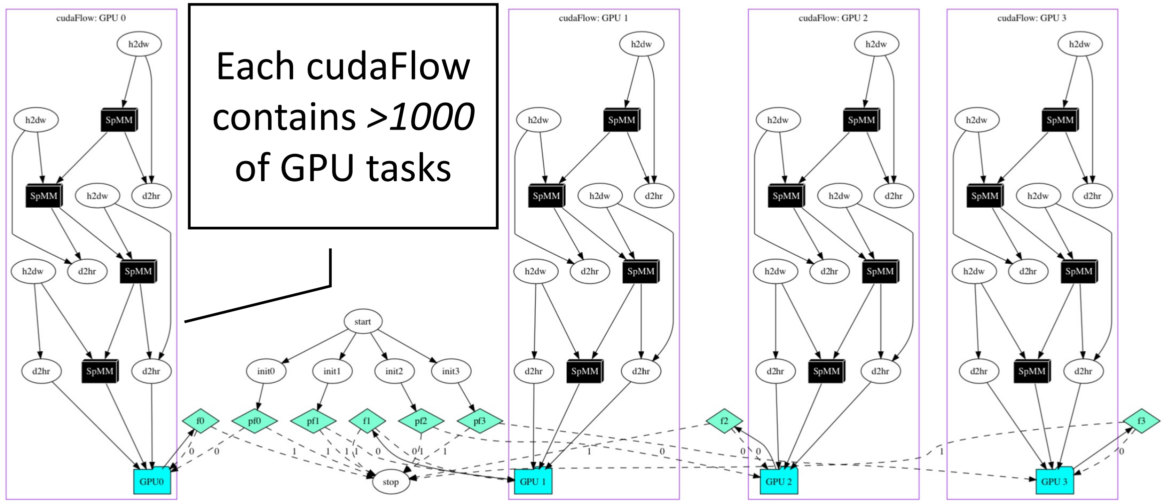
- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

Machine Learning with cudaFlow

- Model neural network inference using cudaFlow
  - Instantiate the CUDA graph once (one-time creation overhead)
  - Iterate inference across data batches on the same executable graph
  - Update graph parameters between successive inference iterations

Radix-net neural network





HPEC 2020 Sparse Neural Network Inference Graph Challenge: https://graphchallenge.mit.edu/champions

Machine Learning with cudaFlow

- Our method "SNIG" *
- Baseline
  - Google's method "Gpipe"
  - Nvidia's method "BF"
- Neural networks
  - Four neuron numbers
    - 1024, 4096, 16384, 65536
  - Three layer numbers
    - 120, 480, 1920
- 4 RTX 2080 Ti GPUs

		Number of GPUs										
		1		2			3			4		
Neurons	Layers	BF	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG
1024	120	**345.93** (0.682s)	295.28 (0.799s)	576.84 (0.409s)	**589.82** (0.400s)	455.46 (0.518s)	**761.06** (0.310s)	695.95 (0.339s)	689.85 (0.342s)	867.38 (0.272s)	768.50 (0.307s)	**1248.30** (0.189s)
	480	477.83 (1.975s)	**586.52** (1.609s)	801.11 (1.178s)	**1016.93** (0.928s)	926.12 (1.019s)	1061.55 (0.889s)	1273.57 (0.741s)	**1348.16** (0.700s)	1112.87 (0.848s)	1483.83 (0.636s)	**1982.60** (0.476s)
	1920	524.50 (7.197s)	**718.74** (5.252s)	852.50 (4.428s)	**1187.81** (3.178s)	1184.45 (3.187s)	1133.59 (3.330s)	1575.48 (2.396s)	**1647.69** (2.291s)	1220.45 (3.093s)	1876.17 (2.012s)	**2159.53** (1.748s)
4096	120	409.42 (2.305s)	**586.52** (1.609s)	746.02 (1.265s)	934.37 (1.010s)	**980.99** (0.962s)	1106.35 (0.853s)	1053.25 (0.896s)	**1460.86** (0.646s)	1385.78 (0.681s)	1165.08 (0.810s)	**2241.61** (0.421s)
	480	544.55 (6.932s)	**803.84** (4.696s)	962.73 (3.921s)	1376.68 (2.742s)	**1400.69** (2.695s)	1431.50 (2.637s)	1767.26 (2.136s)	**2062.77** (1.830s)	1743.59 (2.165s)	2069.5 (1.824s)	**2761.42** (1.367s)
	1920	586.38 (25.75s)	**867.28** (17.41s)	1032.09 (14.63s)	1551.53 (9.732s)	**1575.48** (9.584s)	1538.09 (9.817s)	2074.67 (7.278s)	**2284.34** (6.610s)	1879.21 (8.035s)	2506.97 (6.023s)	**2948.54** (5.121s)
16384	120	462.32 (8.165s)	**851.53** (4.433s)	881.36 (4.283s)	1290.55 (2.925s)	**1487.34** (2.538s)	1303.47 (2.896s)	1521.51 (2.481s)	**2183.26** (1.729s)	1621.50 (2.328s)	1684.45 (2.241s)	**2914.96** (1.295s)
	480	616.30 (24.50s)	**1076.99** (14.02s)	1137.01 (13.28s)	1887.67 (7.999s)	**1965.31** (7.683s)	1678.28 (8.997s)	2454.80 (6.151s)	**2824.44** (5.346s)	2072.39 (7.286s)	2894.28 (5.217s)	**3736.57** (4.041s)
	1920	663.34 (91.05s)	**1113.94** (54.22s)	1207.71 (50.01s)	2105.92 (28.68s)	**2127.43** (28.39s)	1808.86 (33.39s)	2817.06 (21.44s)	**3022.92** (19.98s)	2230.35 (27.08s)	3412.31 (17.70s)	**3963.12** (15.24s)
65536	120	28.79 (524.3s)	**1021.61** (14.78s)	57.52 (262.5s)	1323.35 (11.41s)	**1870.36** (8.073s)	1332.70 (11.33s)	1486.17 (10.16s)	**2705.51** (5.581s)	1652.74 (9.136s)	1565.85 (9.643s)	**3436.38** (4.394s)
	480	(>1800s)	**1404.60** (43.00s)	58.81 (1027s)	2083.40 (28.99s)	**2583.31** (23.38s)	1817.57 (33.23s)	2768.00 (21.82s)	**3784.33** (15.96s)	2241.94 (26.94s)	3222.94 (18.74s)	**5071.19** (11.91s)
	1920	(>1800s)	**1489.46** (162.2s)	(>1800s)	1501.50 (160.9s)	**2810.51** (85.96s)	1960.97 (123.2s)	1948.32 (124.0s)	**4149.63** (58.22s)	2450.47 (98.59s)	2784.27 (86.77s)	**5561.50** (43.44s)

Bold texts denote the best runtime/throughput results

* Dian-Lun Lin and Tsung-Wei Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," *IEEE High-performance and Extreme Computing Conference (HPEC)*, MA, 2020.

Machine Learning with cudaFlow

- Our method “SNIG” *
- Baseline
  - Google’s method “Gpipe”
  - Nvidia’s method “BF”
- Neural networks
  - Four neuron numbers
    - 1024, 4096, 16384, 65536
  - Three layer numbers
    - 120, 480, 1920
- 4 RTX 2080 Ti GPUs

		Number of GPUs										
		1		2			3			4		
Neurons	Layers	BF	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG
1024	120	**345.93** (0.682s)	295.28 (0.799s)	576.84 (0.409s)	**589.82** (0.400s)	455.46 (0.518s)	**761.06** (0.310s)	695.95 (0.339s)	689.85 (0.342s)	867.38 (0.272s)	768.50 (0.307s)	**1248.30** (0.189s)
	480	477.83 (1.975s)	**586.52** (1.609s)	801.11 (1.178s)	**1016.93** (0.928s)	926.12 (1.019s)	1061.55 (0.889s)	1273.57 (0.741s)	**1348.16** (0.700s)	1112.87 (0.848s)	1483.83 (0.636s)	**1982.60** (0.476s)
	1920	524.50 (7.197s)	**718.74** (5.252s)	852.50 (4.428s)	**1187.81** (3.178s)	1184.45 (3.187s)	1133.59 (3.330s)	1575.48 (2.396s)	**1647.69** (2.291s)	1220.45 (3.093s)	1876.17 (2.012s)	**2159.53** (1.748s)
4096	120	409.42 (2.305s)	**586.52** (1.609s)	746.02 (1.265s)	934.37 (1.010s)	**980.99** (0.962s)	1106.35 (0.853s)	1053.25 (0.896s)	**1460.86** (0.646s)	1385.78 (0.681s)	1165.08 (0.810s)	**2241.61** (0.421s)
	480	544.55 (6.932s)	**803.84** (4.696s)	962.73 (3.921s)	1376.68 (2.742s)	**1400.69** (2.695s)	1431.50 (2.637s)	1767.26 (2.136s)	**2062.77** (1.830s)	1743.59 (2.165s)	2069.5 (1.824s)	**2761.42** (1.367s)
	1920	586.38 (25.75s)	**867.28** (17.41s)	1032.09 (14.63s)	1551.53 (9.732s)	**1575.48** (9.584s)	1538.09 (9.817s)	2074.67 (7.278s)	**2284.34** (6.610s)	1879.21 (8.035s)	2506.97 (6.023s)	**2948.54** (5.121s)
16384	120	462.32 (8.165s)	**851.53** (4.433s)	881.36 (4.283s)	1290.55 (2.925s)	**1487.34** (2.538s)	1303.47 (2.896s)	1521.51 (2.481s)	**2183.26** (1.729s)	1621.50 (2.328s)	1684.45 (2.241s)	**2914.96** (1.295s)
	480	616.30 (24.50s)	**1076.99** (14.02s)	1137.01 (13.28s)	1887.67 (7.999s)	**1965.31** (7.683s)	1678.28 (8.997s)	2454.80 (6.151s)	**2824.44** (5.346s)	2072.39 (7.286s)	2894.28 (5.217s)	**3736.57** (4.041s)
	1920	663.34 (91.05s)	**1113.94** (54.22s)	1207.71 (50.01s)	2105.92 (28.68s)	**2127.43** (28.39s)	1808.86 (33.39s)	2817.06 (21.44s)	**3022.92** (19.98s)	2230.35 (27.08s)	3412.31 (17.70s)	**3963.12** (15.24s)
65536	120	28.79 (524.3s)	**1021.61** (14.78s)	57.52 (262.5s)	1323.35 (11.41s)	**1870.36** (8.073s)	1332.70 (11.33s)	1486.17 (10.16s)	**2705.51** (5.581s)	1652.74 (9.136s)	1565.85 (9.643s)	**3436.38** (4.394s)
	480	(>1800s)	**1404.60** (43.00s)	58.81 (1027s)	2083.40 (28.99s)	**2583.31** (23.38s)	1817.57 (33.23s)	2768.00 (21.82s)	**3784.33** (15.96s)	2241.94 (26.94s)	3222.94 (18.74s)	**5071.19** (11.91s)
	1920	(>1800s)	**1489.46** (162.2s)	(>1800s)	1501.50 (160.9s)	**2810.51** (85.96s)	1960.97 (123.2s)	1948.32 (124.0s)	**4149.63** (58.22s)	2450.47 (98.59s)	2784.27 (86.77s)	**5561.50** (43.44s)

Bold texts denote the best runtime/throughput results

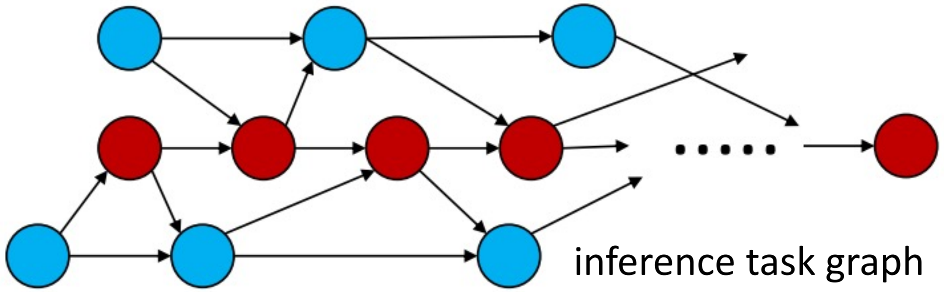
> 2x faster

* Dian-Lun Lin and Tsung-Wei Huang, “A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism,” *IEEE High-performance and Extreme Computing Conference (HPEC)*, MA, 2020.

Machine Learning with cudaFlow Capturer

- Model neural network inference using cudaFlow Capturer *

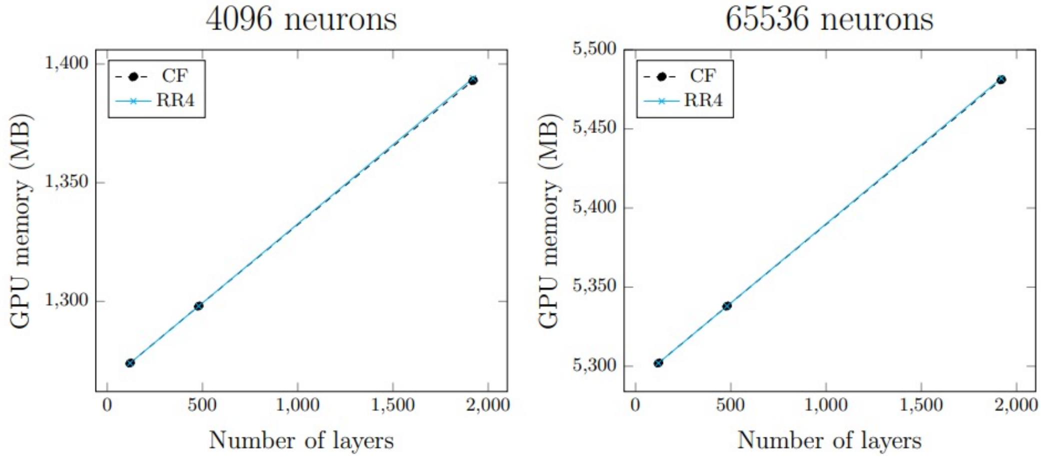
Neurons/Layers	120	480	1920	Model Size	Image Nonzeros
4096	599	2399	9599	5.40 GB	25,019,051
65536	599	2399	9599	94.70 GB	392,191,985



inference task graph

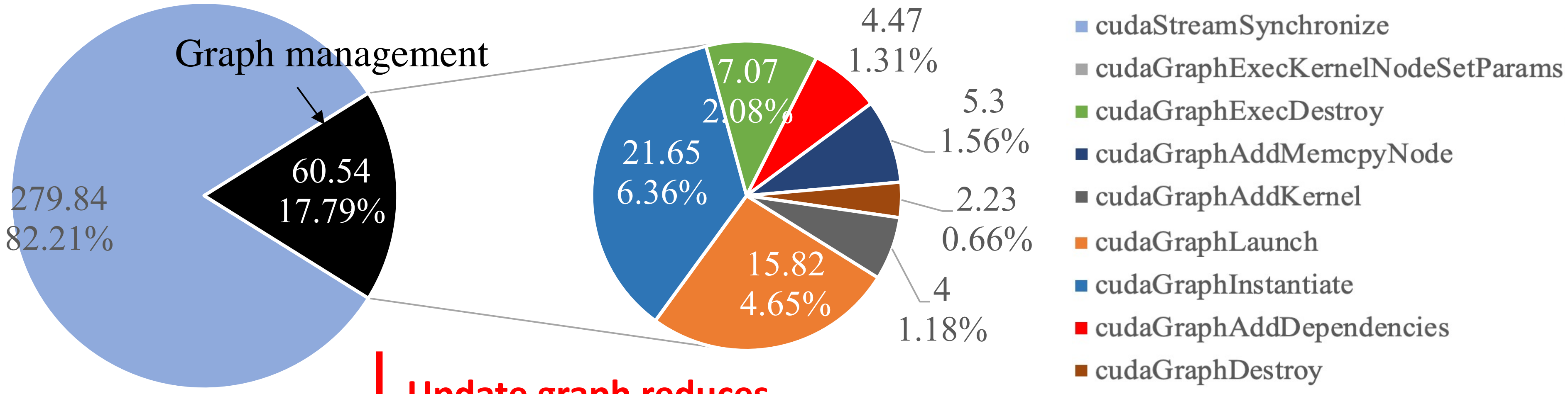
Runtime

#Neurons	#Layers	cudaFlow	cudaFlowCapturer			
			RR1	RR2	RR4	RR8
4096	120	1.61	1.34	1.19	1.20	1.19
	480	4.70	4.74	4.19	4.19	4.20
	1920	17.41	19.14	17.08	17.14	17.15
65536	120	14.78	15.99	14.06	14.06	14.05
	480	43.00	50.59	42.92	42.81	42.90
	1920	162.20	193.11	162.12	162.35	162.30

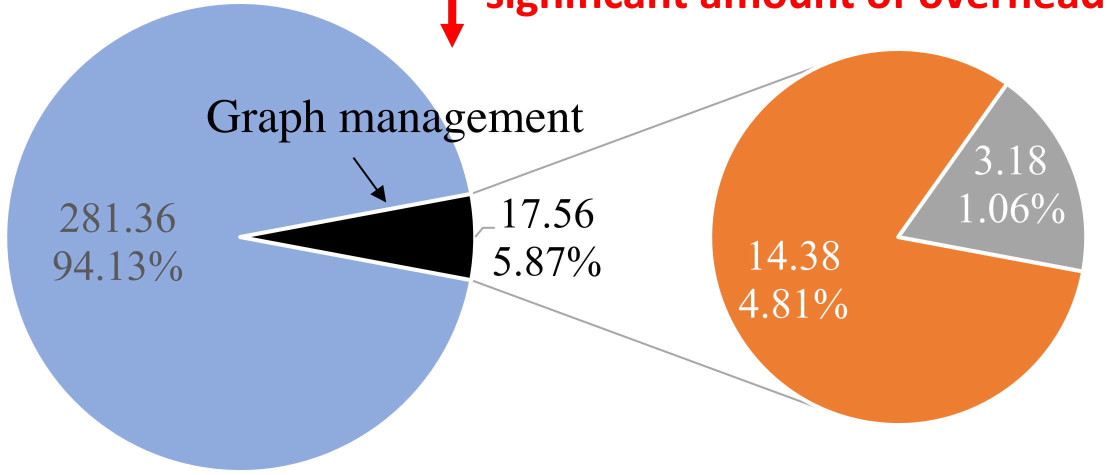


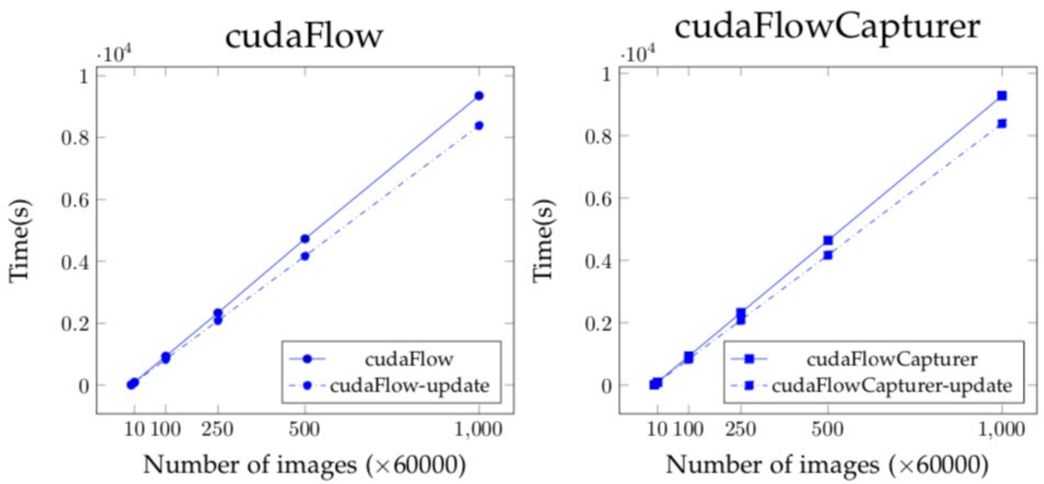
* Dian-Lun Lin and Tsung-Wei Huang, "Efficient GPU Computation using Task Graph Parallelism," *European Conference on Parallel and Distributed Computing (Euro-Par)*, Portugal, 2021

Machine Learning with cudaFlow Update



Update graph reduces significant amount of overhead

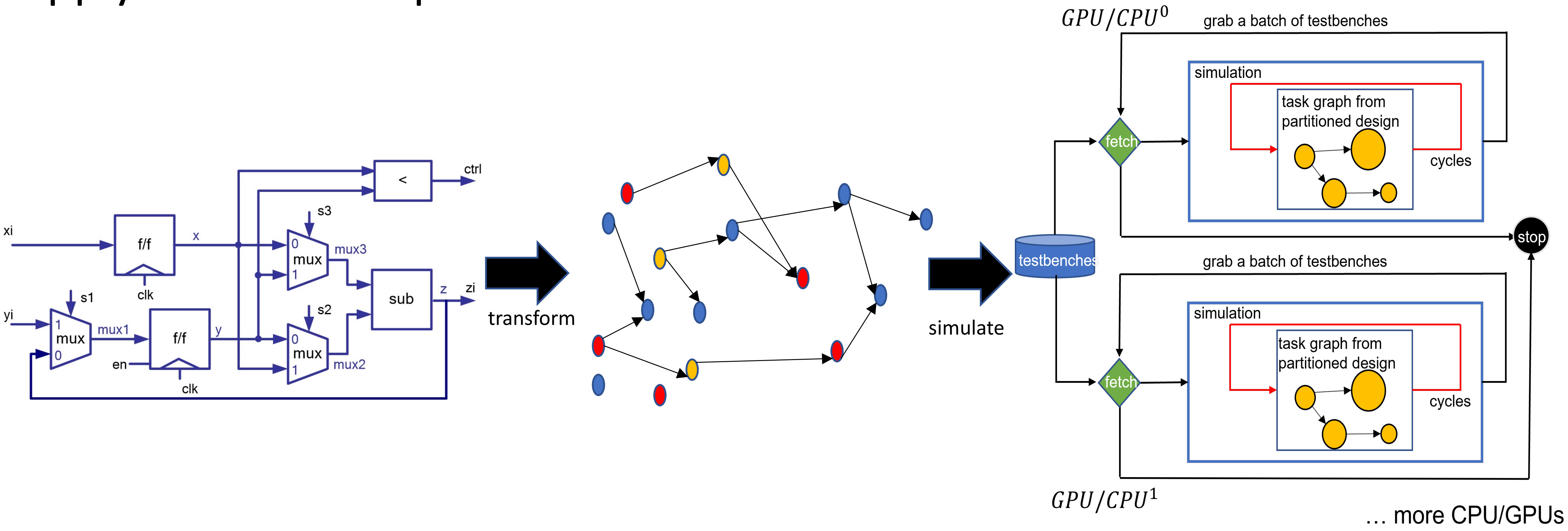




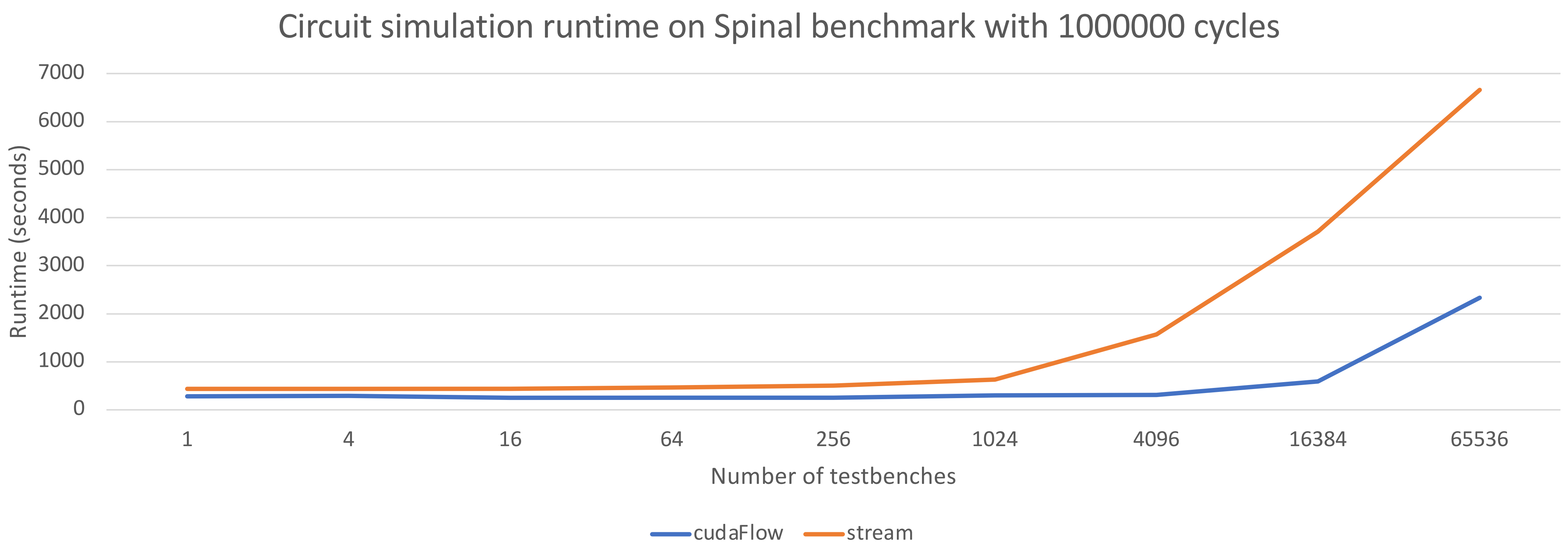
Circuit Simulation

Transform a hardware design into a task graph

Apply cudaFlow to perform circuit simulation



Circuit Simulation (cont’d)



Agenda

- Understand the motivation behind cudaFlow
- Learn to use the cudaFlow C++ programming model
- Dive into the cudaFlow transformation algorithm
- Evaluate cudaFlow on real-world large GPU applications
- Conclusion

Conclusion

- We have presented the motivation behind cudaFlow
- We have presented the cudaFlow C++ programming model
  - Explicit graph construction using cudaFlow
  - Implicit graph capturing using cudaFlowCapturer
  - Integration to the Taskflow project: https://taskflow.github.io
- We have presented the cudaFlow transformation algorithm
- We have presented the performance of cudaFlow
  - Large-scale machine learning workload
  - Large-scale circuit simulation workload
- Future work will focus on integrating coroutine into cudaFlow

Thank You All Using cudaFlow/Taskflow!

Use the right tool for the right job

Taskflow: https://taskflow.github.io

Thank You