CS 3580 - Advanced Topics in Parallel Computing

Inter-Block GPU Communication via Fast Barrier Synchronization

Mohammad Hasanzadeh-Mofrad
University of Pittsburgh
September 12, 2017

General Purpose Graphics Processing Unit (GPGPU)

- Dedicated memory
  - Graphics Double Data Rate (GDDR)
- Interface with the motherboard using:
  - Peripheral Component Interconnect (PCI)
  - PCI Express (PCIe)
  - Accelerated Graphics Port (AGP)
- Application
  - Parallel computing
  - Virtual Realty
  - Security
  - Artificial Intelligence
  - Machine Learning

https://en.wikipedia.org/wiki/Graphics_processing_unit

War of Gods (among desktop GPUs)

NVIDIA

- **Nvidia Titan X**[1]
- NVIDIA CUDA® Cores: 3840
- Clock: 1582 MHz
- Memory Speed: 11.4 GBps
- Memory Amount: 12 GB GDDR5X
- Memory Interface Width: 384-bit
- Memory Bandwidth: 547.7 GB/s
- Price: $1,200.^00^

AMD

- **R9 Fury X**[2]
- Stream Processor Units: 4096
- Clock Speed: 1050 MHz
- Memory Speed: 12.5 GBps
- Memory Amount: 4GB GDDR5
- Memory Interface Width: 350-bit
- Memory Bandwidth: 512 GB/S
- Price: $900.^00^

1. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/
2. http://www.amd.com/en-us/products/graphics/desktop/r9

GPU architecture break down

- Thread-grade
- **CUDA kernel grid**
    - CUDA thread block
        - CUDA thread

- Core-grade
- **CUDA-enabled GPU**
    - Texture Processing Cluster (TPC)
        - Streaming Multiprocessor (SM)
            - Streaming Processor (SP)
                - Floating Point Unit (FPU)

http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Article information

Conference: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)

h5-index: 43

Title: Inter-block GPU communication via fast barrier synchronization

Authors: Shucai Xiao and Wu-chun Feng

Affiliation: Virginia Tech

Citation (as of September 2017): 213

https://scholar.google.com/citations?view_op=top_venues

Gist of the idea

- **GPUs** suits well on parallel application with **minimal inter-block communication** because there is not any inter-block communication on the GPU itself.
- **Inter-block communication** is necessary when multiple thread blocks require to communicate with each other.
- It is implemented using global memory and barrier synchronization across blocks which is called **CPU synchronization**.
- Currently this process is only available via **CPU** which incur significant overhead.
- Since GPUs lack explicit support for inter-block communication, Shucai Xiao and Wu-chun Feng propose two approaches:
  - GPU lock-based synchronization
  - GPU lock-free synchronization

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Raise an objection

- "Today, improving the computational capability of a processor comes from increasing its number of processing cores rather than increasing its clock speed."
- I'm afraid the above statement is false because
  - Intel Nehalem microarchitecture-based (1st generation)
    - Intel Core i7 – Extreme Edition comes with
      - **6** cores
      - **3.2** GHz clock rate
      - in January **2010**
  - Intel Kaby Lake microarchitecture (7th generation)
    - Intel Core i7 – Performance comes with
      - **4/8** cores/threads
      - **4.4** GHz clock rate
      - in January **2017**

1. https://en.wikipedia.org/wiki/Intel_Core

Kernel execution time on the GPU[1]

- Kernel execution time consist of three phases
  1. Kernel **launch** to the GPU (t_O)
  2. **Computation** on the GPU (t_C)
  3. Inter-block GPU **communication** via barrier synchronization (t_S)
- where for *M* kernel launches

$$T = \sum_{i=1}^{M} (t_O^i + t_C^i + t_S^i)$$

- Factors contribute to these three time components
  1. t_O: Data transfer rate, Kernel code size.
  2. t_C: memory access method, thread organization.
  3. t_S: Synchronization parameters.

1. Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

So, what's the incentive?

- Result for the percentage of time spend on inter-block communication for three benchmarks is

FFT	SWat*	Bitonic Sort
17.8%	49.2%	59.6%

- Incentive: In contrast to previous work which focuses on optimizing GPU computation (phase 2), the authors focus on GPU synchronization (phase 3).

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.
*Smith-Waterman Algorithm (SWat)

CUDA Synchronization[1]

- **__syncthreads()**[2] method is a block level synchronization barrier implemented in CUDA programming model which enables **intra-block communication** via shared memory or global memory.
- However, there is no explicit support for communication across different blocks i.e. **inter-block communication** in CUDA programming model.
- Thus, **__syncthreads()** synchronize threads across blocks and not the grid.
- Again reaching the same incentive for synchronizing *M* kernel launches:

Kernel launch t_O^1	Computation t_C^1	Synchronization t_S^1	Kernel launch t_O^2	Computation t_C^2	Synchronization t_S^2	……

1. Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.
2. Feng, Wu-chun, and Shucai Xiao. “To GPU synchronize or not GPU synchronize?." ISCAS, 2010.

Explicit/Implicit Synchronization[1]

- **CPU explicit synchronization** using **cudaThreadSynchronize()**[2]

$$\mathrm{T} = \sum_{i=1}^{M} (t_O^i + t_C^i + t_S^i)$$

```
for()
{
   __kernel_func<<<grid, block>>>();
   cudaThreadSynchronize();
}
```

- **CPU implicit synchronization** without **cudaThreadSynchronize()**[2]

$$\mathrm{T} = t_O^1 + \sum_{i=1}^{M} (t_C^i + t_{CIS}^i)$$

```
for()
{
   __kernel_func<<<grid, block>>>();
   // Without cudaThreadSynchronize();
}
```

- Multiple kernel launches, time can be overlapped by previous kernels.

1. Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.
2. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

GPU Synchronization

- In GPU synchronization, a kernel is launched only once.
- Instead of relaunching a kernel, a **barrier function __gpu_sync()** is called.
- Here, the __*device_func()* implements the behavior of __*kernel_func(),* but it is a *device function* instead of a *global function*. So, it is called on the *device* rather than the *host*.

```
__global__ void __kernel_func1()
{
   for()
   {
      __device_func();
      __gpu_sync();
   }
}
```

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU Synchronization Time

- The kernel execution time in the GPU synchronization is

$$\mathrm{T} = t_O + \sum_{i=1}^{M} (t_C^i + t_{GS}^i)$$

- where M is number of barriers needed for the kernel execution, t_O is the kernel launch time, $t^i{}_C$ is the computation time, and $t^i{}_{GS}$ is the synchronization time.

Amdahl's law

- A program can be split in two parts:
  - parallelizable part *p*
  - Non-parallelizable part 1 – *p*
- Thus, total execution time is:

$$T = (1 - p) + p$$

- According to Amdahl's law:

$$S_T(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

- S_T is the execution speed up
- *N is the number* threads (cores)
- *p* is the parallelizable percentage of the program

https://en.wikipedia.org/wiki/Amdahl%27s_law

How to calculate Computation speedup?

- Kernel execution time for GPU synchronization

$$\mathrm{T} = t_O + \sum_{i=1}^{M} (t_C^i + t_S^i) \,|\, t_S^i = t_{GS}^i$$

- Ignore the kernel launch time

$$\mathrm{T} = \sum_{i=1}^{M} (t_C^i + t_S^i)$$

- Expanding the sum over *i* and absorbing *M*

$$\mathrm{T} = M(t_C + t_S) = Mt_C + Mt_S$$
$$T = t_C + t_S$$

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

How to calculate Computation speedup?

- Thus, the computation time is

$$T = t_C + t_S \rightarrow t_C = T - t_S$$

- Which is the percentage of computation time t_C in the total kernel execution time T.
- So, if only computation is accelerated:

$$S_T = \frac{1}{\left(1 - \frac{t_C}{T}\right) + \frac{\frac{t_C}{T}}{S_C}}$$

- If we consider the computation as the parallel percentage, then

$$p = \frac{t_C}{T}$$

- Thus

$$S_T = \frac{1}{(1 - p) + \frac{p}{S_C}}$$

- S_T is the kernel execution speedup gained by reducing the computation time
- S_C is the computation speed up.

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Amdahl's law – What if …?

- What if we try to gain speedup by parallelizing the non-parallelizable part?
- A program can be split in two parts:
  - parallelizable part *p*
  - Non-parallelizable part 1 – *p*
- But now
  - parallelizable part *p* ➔ Non-parallelizable part (1 – *p*)
  - Non-parallelizable part 1 – *p* ➔ parallelizable part 1 – (1 – *p*) = *p*

- Thus

$$S_T(N) = \frac{1}{p + \frac{(1-p)}{N}}$$

- S_T is the execution speed up
- *N is the number* threads (cores)
- *p* is the parallelizable percentage of the program

https://en.wikipedia.org/wiki/Amdahl%27s_law

How to calculate GPU synchronization speedup?

- So, if the synchronization time is reduced, according to Amdahl's law, the maximum kernel execution speedup is constrained by

$$S_T = \frac{1}{\left(\frac{t_C}{T}\right) + \frac{(1 - \frac{t_C}{T})}{S_S}} = \frac{1}{p + \frac{(1-p)}{S_S}}$$

- Where S_T is the kernel execution speedup gained by reducing the synchronization time
- $p = \frac{t_C}{T}$ is the percentage of the computation time t_C in the total kernel execution time T
- $t_S = T - t_C$ synchronization time of the CPU

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Possible maximum accelerated speedup

- Computation speedup

$$S_T = \frac{1}{(1 - p) + \frac{p}{S_C}}$$

$$p = \frac{t_C}{T}; t_C = T - t_S$$

- The larger the p is, the more speedup can be gained with a fixed S_C

- Synchronization speedup

$$S_T = \frac{1}{p + \frac{(1-p)}{S_S}}$$

$$p = \frac{t_C}{T};\ t_S = T - t_C$$

- The smaller the p is, the more speedup can be gained with a fixed S_S

Algorithm	FFT	SWat	Bitonic Sort
p	0.82	0.51	0.40
Possible Max Speedup with only computation accelerated	5.61	2.03	1.68

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU Synchronization strategies

1. **GPU lock-based synchronization**
    - A mutex + a CUDA atomic operation
2. GPU lock-free synchronization
    - A lock-free algorithm

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU lock-based Synchronization

- Implementing barrier function **__gpu_sync()**
  1. After a block completes its computation
  2. One of its threads (**leading thread**) will increment *g_mutex*
  3. And compare *g_mutex* to *goalVal*
  4. If g_mutex is equal to goalVal, the synchronization is completed and each thread block can proceed to the next stage of computation

1 2 3 4

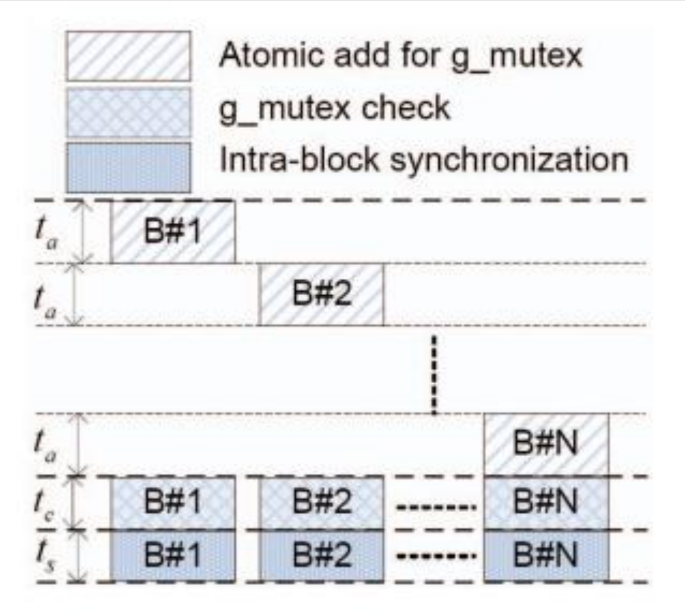
```
__device__ volatile int g_mutex; //the mutex variable
//GPU lock-based synchronization function
__device__ void __gpu_sync(int goalVal){
  //thread ID in a block
  int tid_in_block = threadIdx.x * blockDim.y + threadIdx.y;
  // only thread 0 is used for synchronization
  if (tid_in_block == 0){
    atomicAdd((int *)&g_mutex, 1);
    //only when all blocks add 1 to g_mutex
    //will g_mutex equal to goalVal
    while(g_mutex != goalVal){
      //Do nothing here
    }
  }
  __syncthreads();
}
```

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU lock-based Synchronization

- __gpu_sync() execution time consists of
  1. Atomic addition of *g_mutex* which can be only executed sequentially by different blocks $\boldsymbol{t_a}$
  2. Busy waiting for *g_mutex* checking which can be executed in parallel $\boldsymbol{t_c}$
  3. Synchronization of threads within a block via **__synchthreads()** which also cane be executed in parallel $\boldsymbol{t_s}$
- Thus for N blocks

$$t_{GBS} = N * t_a + t_c + t_s$$



Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU Synchronization strategies

1. GPU lock-based synchronization
   - A mutex + a CUDA atomic operation
2. **GPU lock-free synchronization**
   - A lock-free algorithm

GPU Lock-free Synchronization

- The **atomic add operation** of *lock-based synchronization* on *g_mutex* is a performance bottleneck because it is executed sequentially by each thread block.
- In *lock-free synchronization*, there is **no atomic operation** and all the operations can be executed in parallel.
- Synchronization of different thread blocks is controlled by threads in a single block, which can be synchronized by calling _synchthreads().
- The lock-free synchronization strategy uses two arrays ***Arrayin*** and ***Arrayout*** to coordinate the synchronization requests from various blocks where thread block *i* is mapped to the i_{th} element.

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

GPU Lock-free Synchronization (algorithm)

1. When the i_{th} block is ready for communication

    1.1 Its leading thread sets the i_{th} element of *Arrayin* to *goalVal*

    1.2 busy wait on i_{th} element of *Arrayout* to be set to *goalVal*.

1.1

1.2

```
//GPU lock-free synchronization function
__device__ void __gpu_sync(int goalVal, volatile int *Arrayin, volatile int *Arrayout){
    // thread ID in a block
    int tid_in_blk = threadIdx.x * blockDim.y+ threadIdx.y;
    int nBlockNum = gridDim.x * gridDim.y;
    int bid = blockIdx.x * gridDim.y + blockIdx.y;
    // only thread 0 is used for synchronization
    if (tid_in_blk == 0) {
        Arrayin[bid] = goalVal;
    }
    if (bid == 1) {
        if (tid_in_blk < nBlockNum) {
            while (Arrayin[tid_in_blk] != goalVal){
                //Do nothing here
            }
        }
        __syncthreads();
        if (tid_in_blk < nBlockNum) {
            Arrayout[tid_in_blk] = goalVal;
        }
    }
    if (tid_in_blk == 0) {
        while (Arrayout[bid] != goalVal) {
            //Do nothing here
        }
    }
    __syncthreads();
}
```

GPU Lock-free Synchronization (algorithm)

2. In parallel

    2.1. the first *N* threads in block 1 repeatedly check if all elements in *Arrayin* are equal to *golaVal*, with i_{th} of 1st block checking i_{th} element of *Arrayin*.

    2.2 After all elements in *Arrayin* are set to *goalVal*

    2.3 The *__synchreads()* is called

    2.4 Each checking thread *i* sets the i_{th} element of *Arrayout* to *goalVal*.

2.1

2.2

2.3

2.4

```
//GPU lock-free synchronization function
__device__ void __gpu_sync(int goalVal, volatile int *Arrayin, volatile int *Arrayout){
    // thread ID in a block
    int tid_in_blk = threadIdx.x * blockDim.y+ threadIdx.y;
    int nBlockNum = gridDim.x * gridDim.y;
    int bid = blockIdx.x * gridDim.y + blockIdx.y;
    // only thread 0 is used for synchronization
    if (tid_in_blk == 0) {
        Arrayin[bid] = goalVal;
    }
    if (bid == 1) {
        if (tid_in_blk < nBlockNum) {
            while (Arrayin[tid_in_blk] != goalVal){
                //Do nothing here
            }
        }
        __syncthreads();
        if (tid_in_blk < nBlockNum) {
            Arrayout[tid_in_blk] = goalVal;
        }
    }
    if (tid_in_blk == 0) {
        while (Arrayout[bid] != goalVal) {
            //Do nothing here
        }
    }
    __syncthreads();
}
```

GPU Lock-free Synchronization (algorithm)

3. i_{th} block will continue execution

   3.1 once its leading thread sees the i_{th} element of *Arrayout* is set to *goalVal.*

3.1

```
//GPU lock-free synchronization function
__device__ void __gpu_sync(int goalVal, volatile int *Arrayin, volatile int *Arrayout){
    // thread ID in a block
    int tid_in_blk = threadIdx.x * blockDim.y+ threadIdx.y;
    int nBlockNum = gridDim.x * gridDim.y;
    int bid = blockIdx.x * gridDim.y + blockIdx.y;
    // only thread 0 is used for synchronization
    if (tid_in_blk == 0) {
        Arrayin[bid] = goalVal;
    }
    if (bid == 1) {
        if (tid_in_blk < nBlockNum) {
            while (Arrayin[tid_in_blk] != goalVal){
                //Do nothing here
            }
        }
        __syncthreads();
        if (tid_in_blk < nBlockNum) {
            Arrayout[tid_in_blk] = goalVal;
        }
    }
    if (tid_in_blk == 0) {
        while (Arrayout[bid] != goalVal) {
            //Do nothing here
        }
    }
    __syncthreads();
}
```

GPU Lock-free Synchronization

- Execution time of __gpu_sync() is

$$t_{GFS} = t_{SI} + t_{CI} + 2t_s + t_{SO} + t_{CO}$$

- t_{SI}, time for setting an element in *Arrayin*
- t_{CI}, time for checking an element in *Arrayin*
- t_s, time for intra-block synchronization
- t_{SO}, time for setting an element in *Arrayout*
- t_{CO}, time for checking an element in *Arrayout*

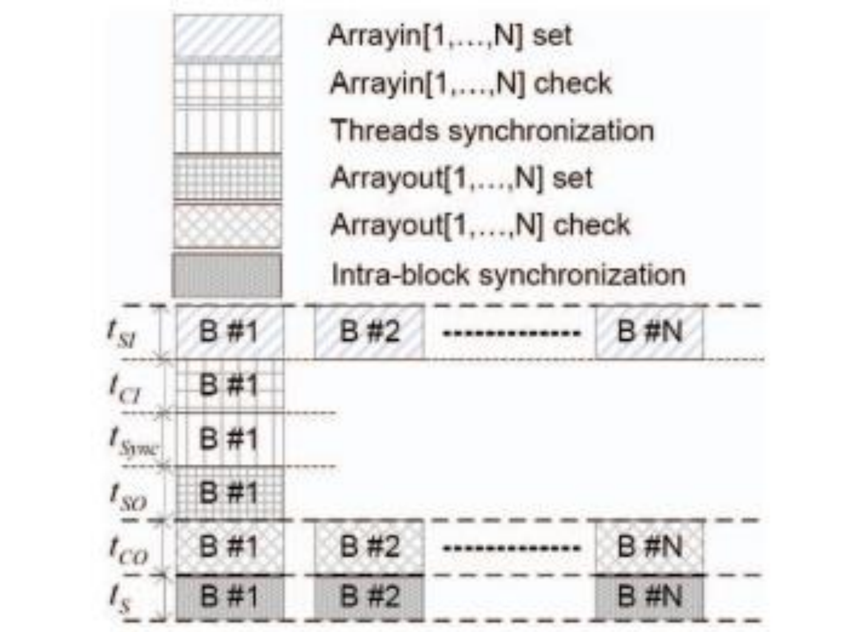
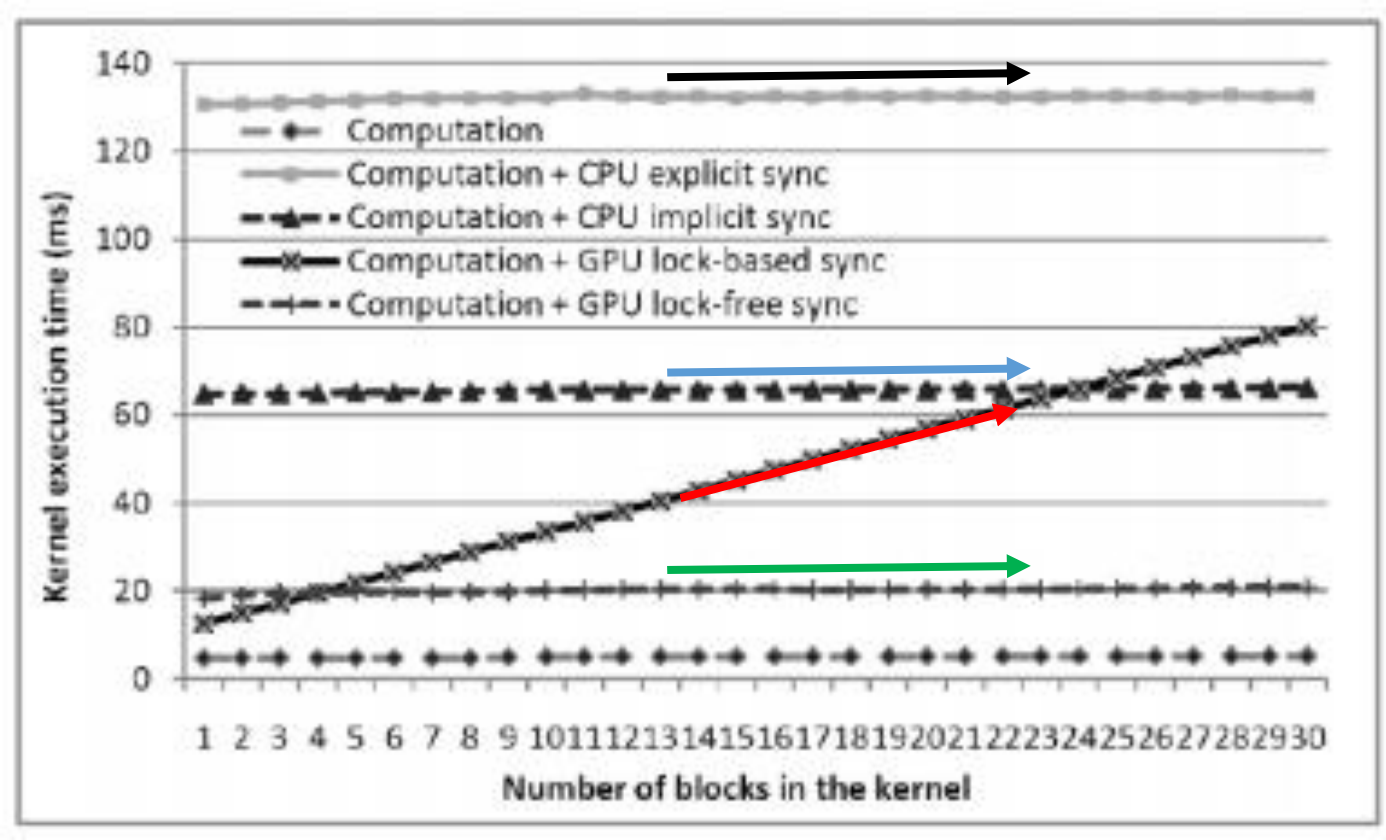


Fig. 7. Time Composition of GPU Lock-Free Synchronization

- *Note: Kernel execution time is not a function of number of blocks N*

Execution time of the micro-benchmark

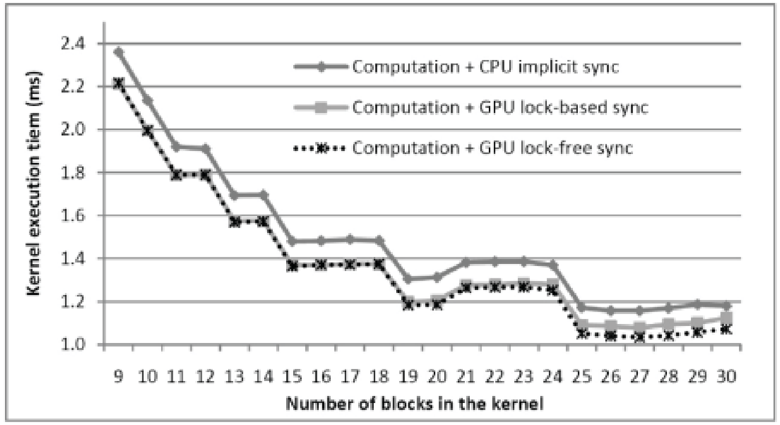
- Micro-benchmark: Compute the mean of two floats for 10,000 times
- CPU explicit synchronization launch a new kernel for each block
- CPU implicit synchronization overlaps the time for kernel launches and pipeline the computation
- GPU lock-based synchronization time is a function a number of blocks in a kernel
- GPU lock-free synchronization does not have an atomic operation and can be executed in parallel which makes the synchronization time almost a constant value



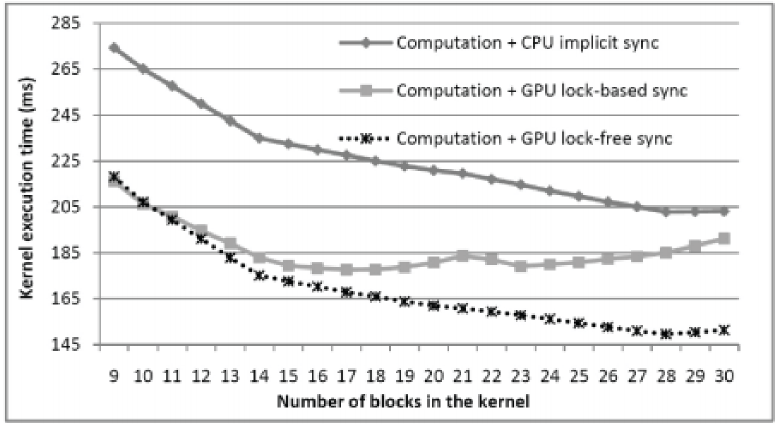
Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Results: Kernel Execution Time

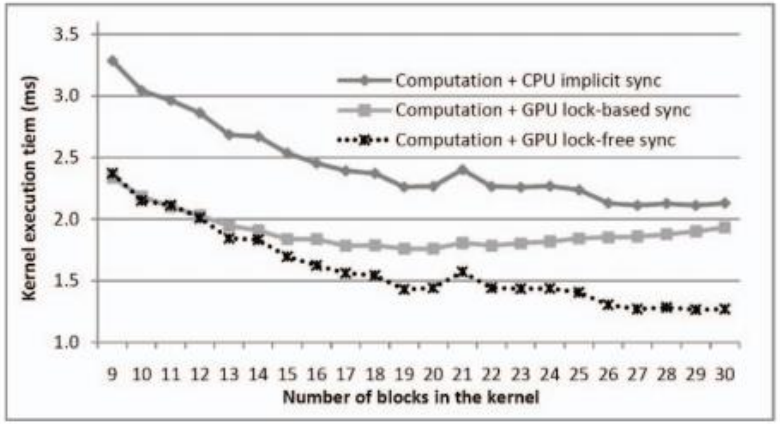
- Test algorithms: FFT, SWat, and Bitonic sort.
- Increasing the number of thread blocks in the kernel, the execution time will be decreased.
  - More resources, more acceleration
- Performance improvement can be seen in all three test algorithms for GPU synchronization strategies with less execution time.



(a) FFT



(b) SWat



(c) Bitonic sort

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Results: Execution Time Speedup

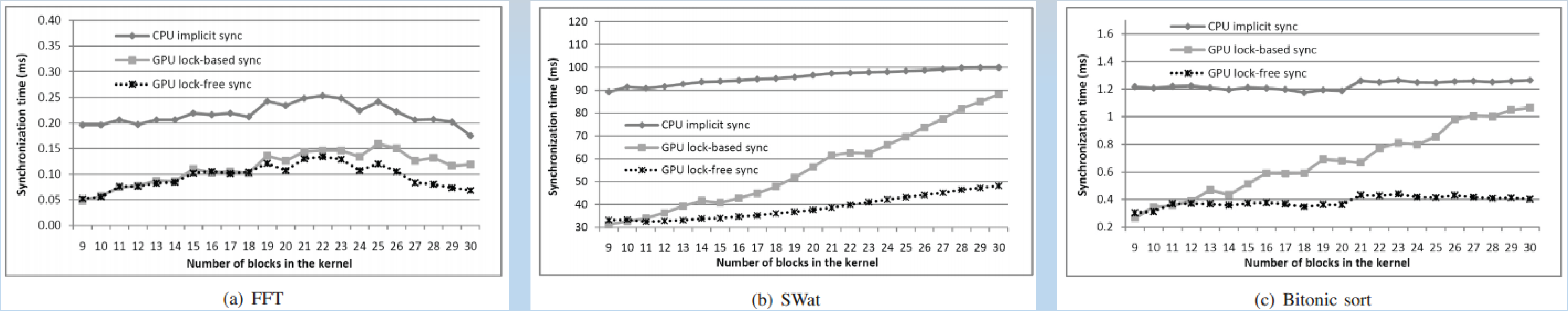
- Gained speedup compare to the sequential implementation of test algorithms.

Algorithm	FFT	SWat	Bitonic sort
Implicit CPU synchronization	62.50	9.53	14.40
GPU lock-based synchronization	67.14	10.89	17.27
GPU lock-free synchronization	69.93	12.93	24.02

Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Results: Synchronization Time

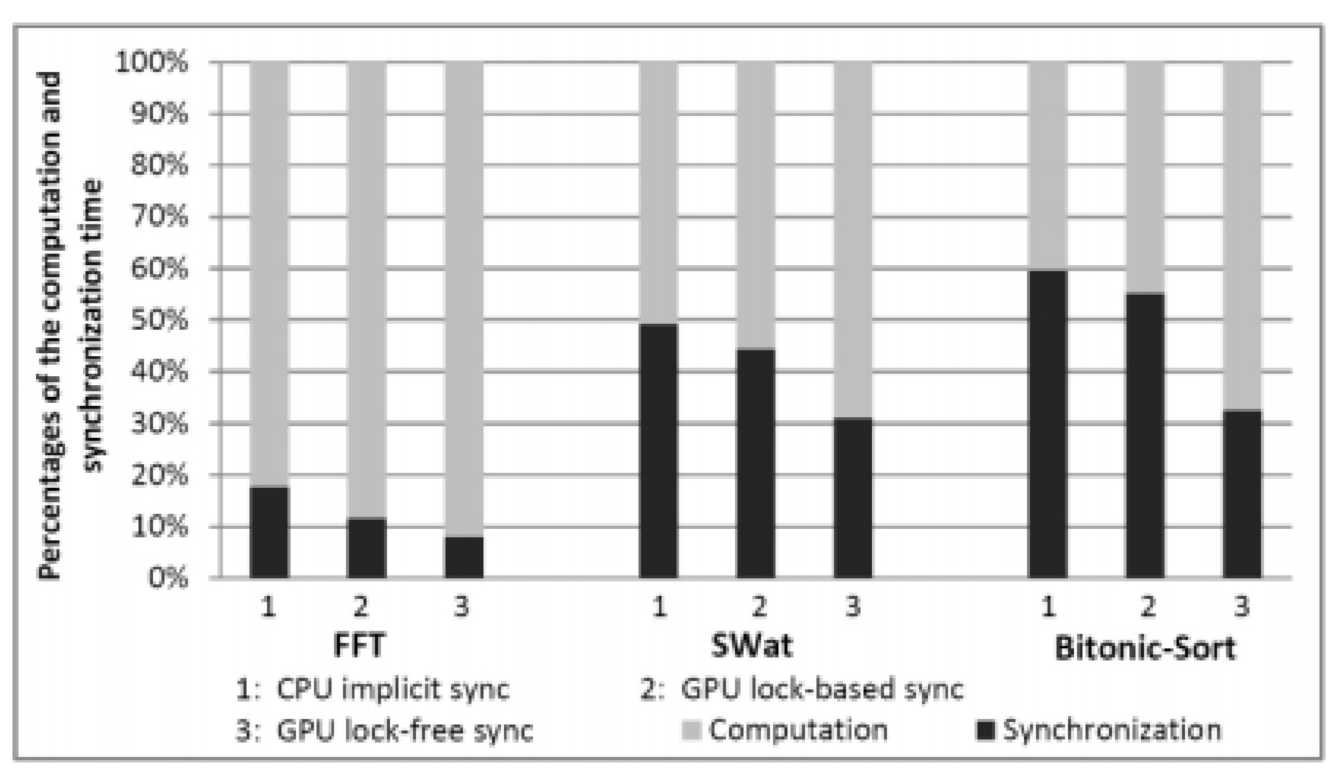
- Roughly, when more blocks configured in the kernel, a little more time is needed for synchronization.
- GPU synchronization strategies are taking less synchronization time compare to implicit CPU synchronization.



Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." IPDPS, 2010.

Percentage of Computation Time and Synchronization Time

- Performance breakdown in percentage of the three test algorithms for different synchronization approaches.
- GPU-based strategies has less synchronization time.



Conclusion

- The literature focuses on optimizing the computation time rather than synchronization time.
- A performance model is proposed:
  - Kernel launch time
  - Kernel execution time
  - Kernel synchronization time
- Two approaches for inter-block communication on GPUs are proposed
  - Lock-based GPU synchronization: Mutex + CUDA atomic operation
  - Lock-free GPU synchronization: Two synchronization arrays
- Results show that proposed GPU synchronization approaches obtained better performance in all test algorithms compare to CPU barrier synchronization.