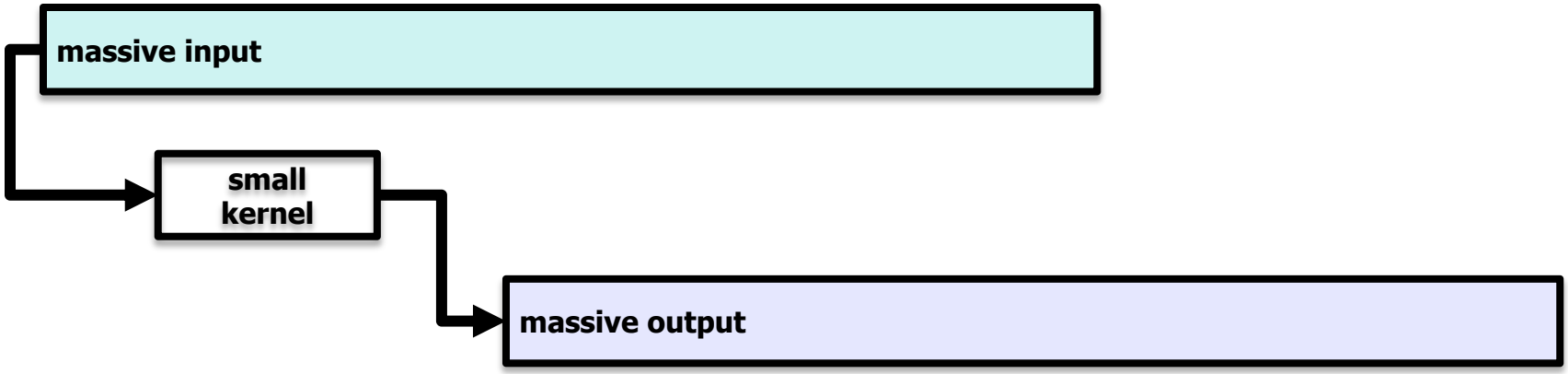
Parallel Patterns: Streaming and AsyncCopy

Prof. Seokin Hong

Streaming Operation

- **problem definition**
  - massive input
  - small kernel
  - massive output



Streaming Operation

- simple example:

```
void kernel(float* dst, float* src, float val, int size) {
    while (size--) {
            *dst++ = *src++ + 10.0F;
    }
}
```

Host Version

```
#include <stdio.h>
#include <stdlib.h>

#define STREAMSIZE 1024
#define GRIDSIZE                (64 * 1024)
#define BLOCKSIZE               512
#define TOTALSIZE               (GRIDSIZE * BLOCKSIZE)

#define     REPEAT              8

void genData(float* ptr, unsigned int size) {
    while (size--) {
            *ptr++ = (float)(rand() % 1000) / 1000.0F;
    }
}
```

Host Version

```
void kernel(float* dst, float* src, float val, int size) {
    while (size--) {
            *dst = 0.0F;
            for (register int j = 0; j < REPEAT; ++j) {
                    *dst += *src;
            }
            dst++;
            src++;;
    }
}
```

Host Version

```
int main(void) {
    float* pSource = NULL;
    float* pResult = NULL;
    int i;
    long long cntStart, cntEnd, freq;
    ……
    // malloc memories on the host-side
    pSource = (float*)malloc(TOTALSIZE * sizeof(float));
    pResult = (float*)malloc(TOTALSIZE * sizeof(float));
    // generate source data
    genData(pSource, TOTALSIZE);
    ….
```

Host Version

```
    // perform the action
    kernel(pResult, pSource, SHIFT, TOTALSIZE);
    ….
    // print sample cases
    i = 0;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], REPEAT);
    i = TOTALSIZE – 1;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], REPEAT);
    i = TOTALSIZE / 2;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], REPEAT);
    …
}
```

Execution Result

- execution result: for host version

elapsed time = 588533.572312 usec

i= 0: 0.328000 = 0.041000 * 8

i=33554431: 1.856000 = 0.232000 * 8

i=16777216: 2.760000 = 0.345000 * 8

Device Version

```
#include <stdio.h>
#include <stdlib.h>

#define STREAMSIZE     1024
#define GRIDSIZE        (64 * 1024)
#define BLOCKSIZE       512
#define TOTALSIZE       (GRIDSIZE * BLOCKSIZE)

#define      REPEAT       8



void genData(float* ptr, unsigned int size) {
    while (size--) {
            *ptr++ = (float)(rand() % 1000) / 1000.0F;
    }
}
```

Device Version

```
__global__ void kernel(float* dst, float* src) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    dst[i] = 0.0F;
    for (register int j = 0; j < REPEAT; ++j) {
            dst[i] += src[i];
    }
}
```

Device Version

```
int main(void) {
    float* pSource = NULL;
    float* pResult = NULL;
    int i;
    ….
    // malloc memories on the host-side
    pSource = (float*)malloc(TOTALSIZE * sizeof(float));
    pResult = (float*)malloc(TOTALSIZE * sizeof(float));
    // generate source data
    genData(pSource, TOTALSIZE);
    // CUDA: allocate device memory
    float* pSourceDev = NULL;
    float* pResultDev = NULL;
    cudaMalloc((void**)&pSourceDev, TOTALSIZE * sizeof(float));
    cudaMalloc((void**)&pResultDev, TOTALSIZE * sizeof(float));
```

Device Version

```
….
// CUDA: copy from host to device
cudaMemcpy(pSourceDev, pSource, TOTALSIZE * sizeof(float), cudaMemcpyHostToDevice);

// CUDA: launch the kernel
dim3 dimGrid(GRIDSIZE, 1, 1);
dim3 dimBlock(BLOCKSIZE, 1, 1);
kernel<<<dimGrid, dimBlock>>>(pResultDev, pSourceDev);
// CUDA: copy from device to host
cudaMemcpy(pResult, pResultDev, TOTALSIZE * sizeof(float), cudaMemcpyDeviceToHost);
…..
```

Device Version

```
    // print sample cases
    i = 0;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], NUMITER);
    i = TOTALSIZE - 1;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], NUMITER);
    i = TOTALSIZE / 2;
    printf("i=%2d: %f = %f * %d\n", i, pResult[i], pSource[i], NUMITER);
    // CUDA: free the memory
    cudaFree(pSourceDev);
    cudaFree(pResultDev);
    // free the memory
    free(pSource);
    free(pResult);
}
```

Execution Result

- execution result for device version

elapsed time = 108413.183538 usec

i= 0: 0.000001 = 0.041000 * 8

i=33554431: 5.936000 = 0.232000 * 8

i=16777216: 4.920000 = 0.345000 * 8

- elapsed time = 588533.572312 usec (host version)

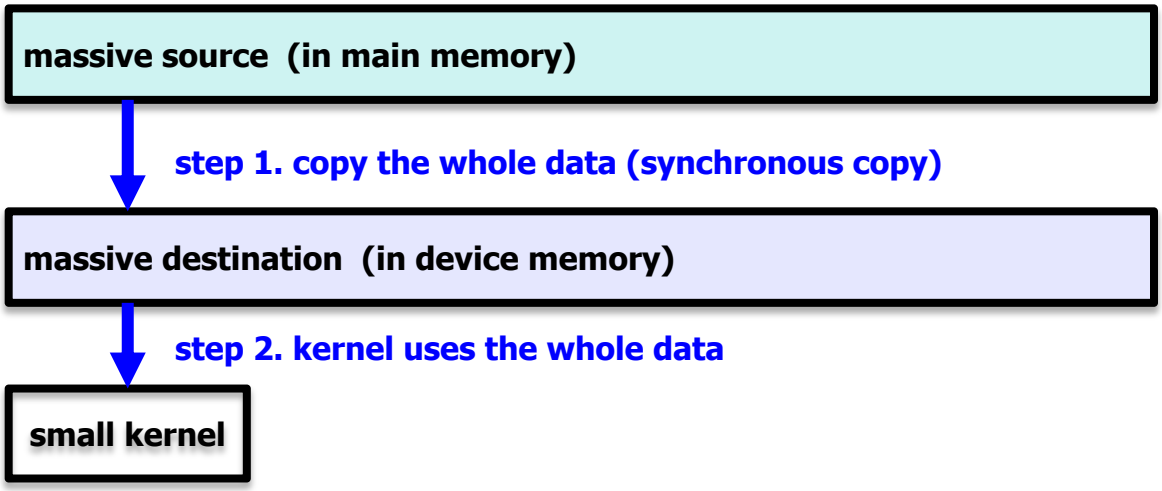
Asynchronous Copy

Big-Size Streaming Operations

- problem definition
  - massive input (may still be being generated !!!)
  - small kernel
  - massive output

Synchronous Streaming Case

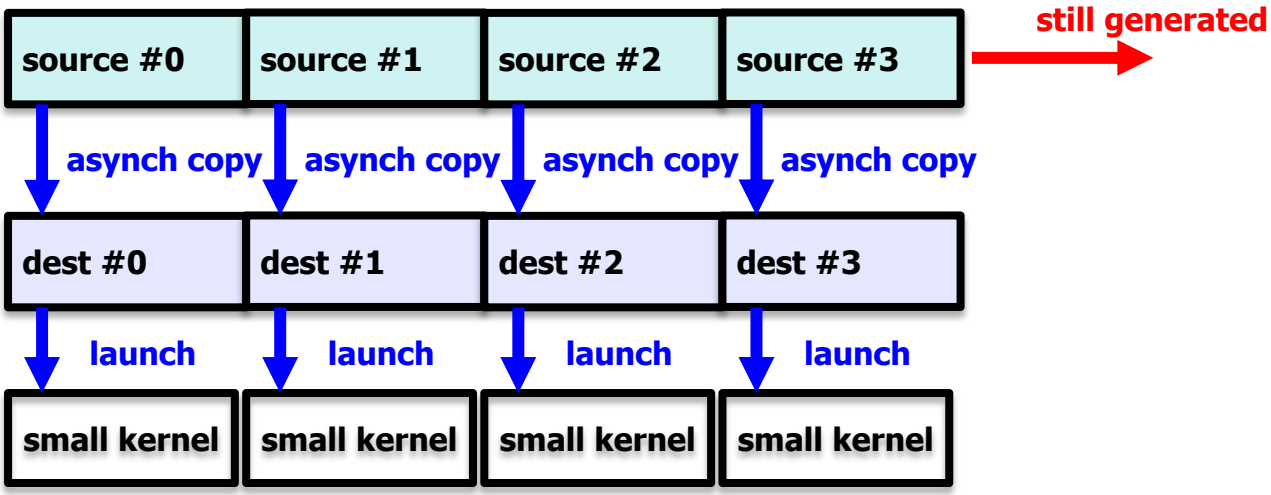
- cudaMemcpy(void* dst, const void* src, size_t count,
  enum kind);
  - kind = cudaMemcpyHostToDevice, …
  - return after the completion of the copy process



Asynchronous Copy

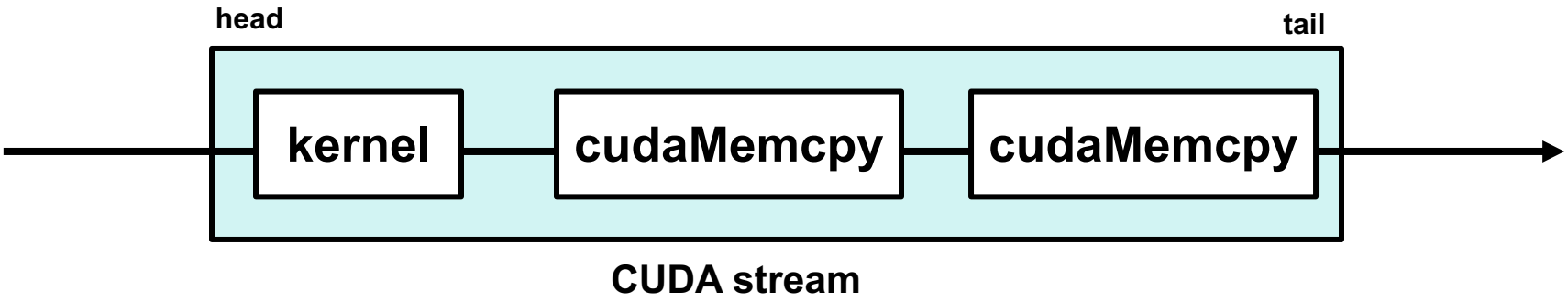
- cudaMemcpyAsync(void* dst, const void* src,
  size_t count, enum kind,
  cudaStream_t stream = 0);
  - return just after starting the copy process

- **Pinned host memory** is required for cudaMemcpyAsync
  - Pinned-memory: Memory that is resident in physical memory pages, and cannot be swapped out, also referred as page-locked
  - cudaMallocHost(), cudaFreeHost()



CUDA streams

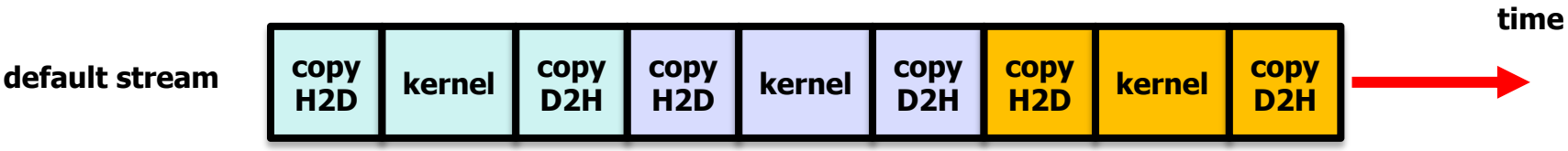
- **A CUDA stream** is an ordered sequence of kernel launches and CUDA runtime API calls that are all executed sequentially, with no overlap (i.e. FIFO queue)
  - Placing a new action at the head of a stream
  - Executing actions from the tail



- **Work in different CUDA streams** can be performed in parallel
- Every kernel launch and CUDA runtime API call is in some stream

CUDA streams

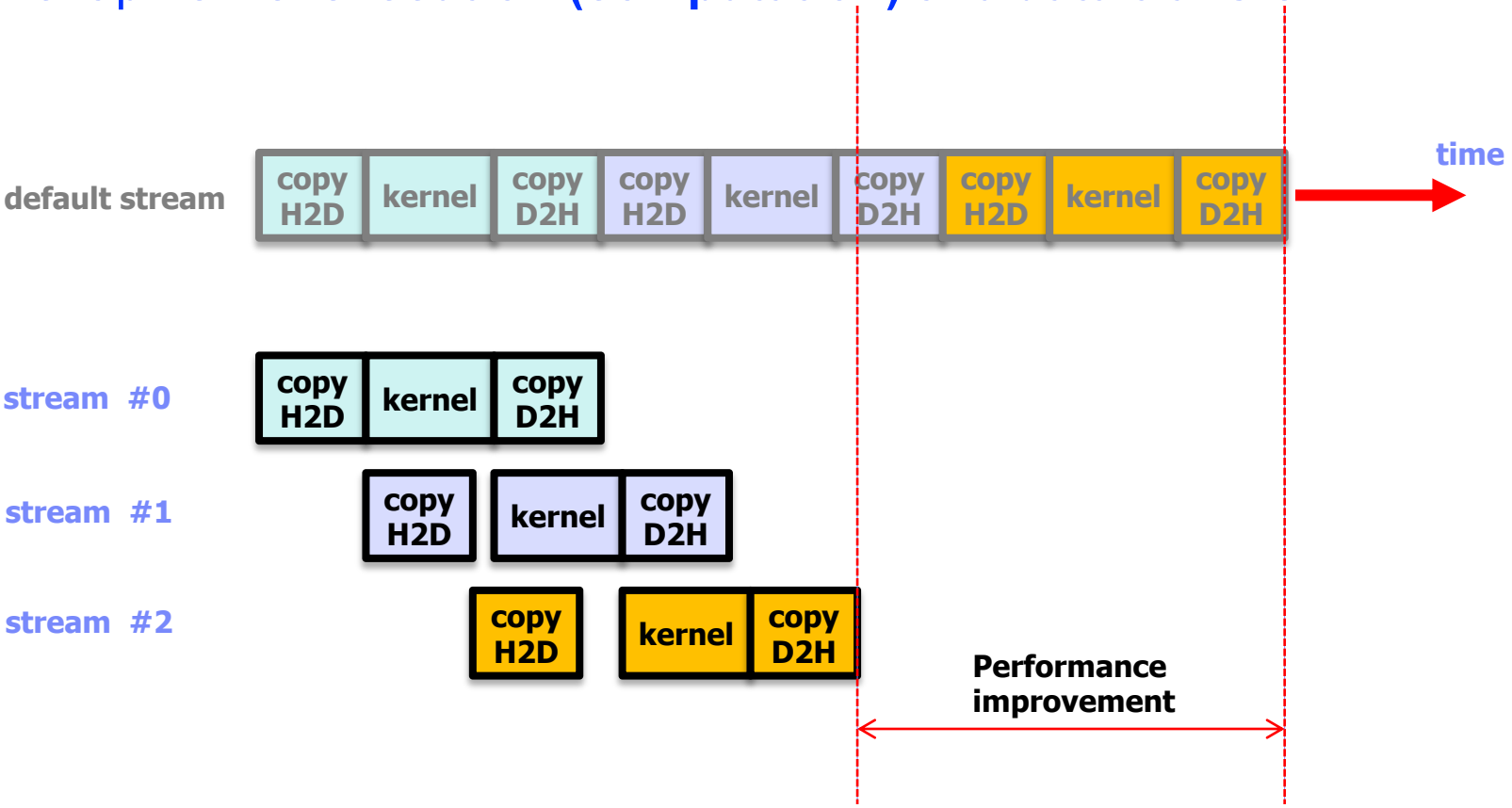
- without explicit streams
  - everything binds to the **default (null) stream**
  - **serial execution** in the default stream



H2D : host to device
D2H : device to host

CUDA streams

- with explicit streams
  - each stream can be **executed in parallel**
  - **copy queue** and **kernel queue** managed independently
  - overlap **kernel execution (computation)** and **data transfer**



Asynchronous Version

```
int main(void) {
    // allocate pinned-memory
    float* pSource = NULL;
    float* pResult = NULL;
    cudaMallocHost((void**)&pSource, TOTALSIZE * sizeof(float));
    cudaMallocHost((void**)&pResult, TOTALSIZE * sizeof(float));
    // create stream object
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    // generate source data
    genData(pSource, TOTALSIZE);
    // CUDA: allocate device memory
    float* pSourceDev = NULL;
    float* pResultDev = NULL;
    cudaMalloc((void**)&pSourceDev, TOTALSIZE * sizeof(float));
    cudaMalloc((void**)&pResultDev, TOTALSIZE * sizeof(float));
```

Asynchronous Version (cont'd)

```
    ....
    // CUDA: copy from host to device
    cudaMemcpyAsync(pSourceDev, pSource, TOTALSIZE * sizeof(float),
                cudaMemcpyHostToDevice, stream);
    // CUDA: launch the kernel
    dim3 dimGrid(GRIDSIZE, 1, 1);
    dim3 dimBlock(BLOCKSIZE, 1, 1);
    kernel<<<dimGrid, dimBlock, 0, stream>>>(pResultDev, pSourceDev);
    // sync to make sure operations complete
    cudaStreamSynchronize(stream);
    // deallocate pinned-memory
    cudaFreeHost(pSource);
    cudaFreeHost(pResult);
    // destroy stream
    cudaStreamDestroy(stream);
    ..
}
```

Multiple Streams

```
int main(void) {
  …
   // create stream object
   cudaStream_t aStream[STREAMSIZE];
   for (i = 0; i < STREAMSIZE; ++i) {
            cudaStreamCreate(&(aStream[i]));
   }

   // CUDA: copy from host to device
   for (i = 0; i < STREAMSIZE; ++i) {
            int offset = TOTALSIZE / STREAMSIZE * i;
            cudaMemcpyAsync(pSourceDev + offset, pSource + offset,
                     TOTALSIZE / STREAMSIZE * sizeof(float),
                     cudaMemcpyHostToDevice, aStream[i]);
   }
  …
```

Multiple Streams (cont'd)

```
// CUDA: launch the kernel
dim3 dimGrid(GRIDSIZE / STREAMSIZE, 1, 1);
dim3 dimBlock(BLOCKSIZE, 1, 1);
for (i = 0; i < STREAMSIZE; ++i) {
        int offset = TOTALSIZE / STREAMSIZE * i;
        kernel<<<dimGrid, dimBlock, 0, aStream[i]>>>(pResultDev + offset,
                                                     pSourceDev + offset);
}

// CUDA: copy from device to host
for (i = 0; i < STREAMSIZE; ++i) {
        cudaMemcpyAsync(pResult + offset, pResultDev + offset,
                        TOTALSIZE / STREAMSIZE * sizeof(float),
                        cudaMemcpyDeviceToHost, aStream[i]);
}
```

Multiple Streams (cont'd)

```
    // CUDA: sync to make sure operations complete and destroy streams
    for (i = 0; i < STREAMSIZE; ++i) {
            cudaStreamSynchronize(aStream[i]);
            cudaStreamDestroy(aStream[i]);
    }
    …
}
```

Elapsed Time Check for CUDA Kernel

- with event type
  - cudaEvent_t

- cudaError_t cudaEventElapedTime(float* msec,
  cudaEvnet_t start, cudaEvent_t end);
  - computes the elapsed time between two events (in milliseconds)

Explicit Synchronization

- cudaDeviceSynchronize(void);
  - Block a host thread until all device operations have completed
- cudaStreamSynchronize(stream);
  - Block a host thread until all operations in a stream have completed
- cudaEventSynchronize(event);
  - Block a host thread until an event is recorded
- cudaStreamWaitEvent(stream, event);
  - Block a stream until an event is recorded

CUDA event functions

- cudaError_t cudaEventCreate(cudaEvent_t* event);
  - create an event object

- cudaError_t cudaEventRecord(cudaEvent_t event,
  cudaStream_t stream);
  - records an event, after all preceding operations

- cudaError_t cudaEventSynchronize(cudaEvent_t event);
  - blocks until the event has actually been recorded

- cudaErrot_t cudaEventDestroy(cudaEvent_t event);
  - destroy the specified event object

Elapsed Time: CUDA Event API

```
cudaEvent_t start, stop; // event object declaration
float time;
cudaEventCreate(&start); // event object create
cudaEventCreate(&stop);

cudaEventRecord(start, 0); // event record
VectorAdd<<<65535,512>>>(dev_A, dev_B, dev_R, size);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop); // synchronization

cudaEventElapsedTime(&time, start, stop); // get the time in msec
cudaEventDestroy(start); // destroy
cudaEventDestroy(stop);

printf("elapsed time = %f msec\n", time);
```