Extending OpenMP map Clause to Bridge Storage and Device Memory

<u>Kewei Yan</u>, Anjia Wang, Xinyao Yi, Yonghong Yan

University of North Carolina at Charlotte, NC, USA

MCHPC'19: Workshop on Memory Centric High Performance Computing

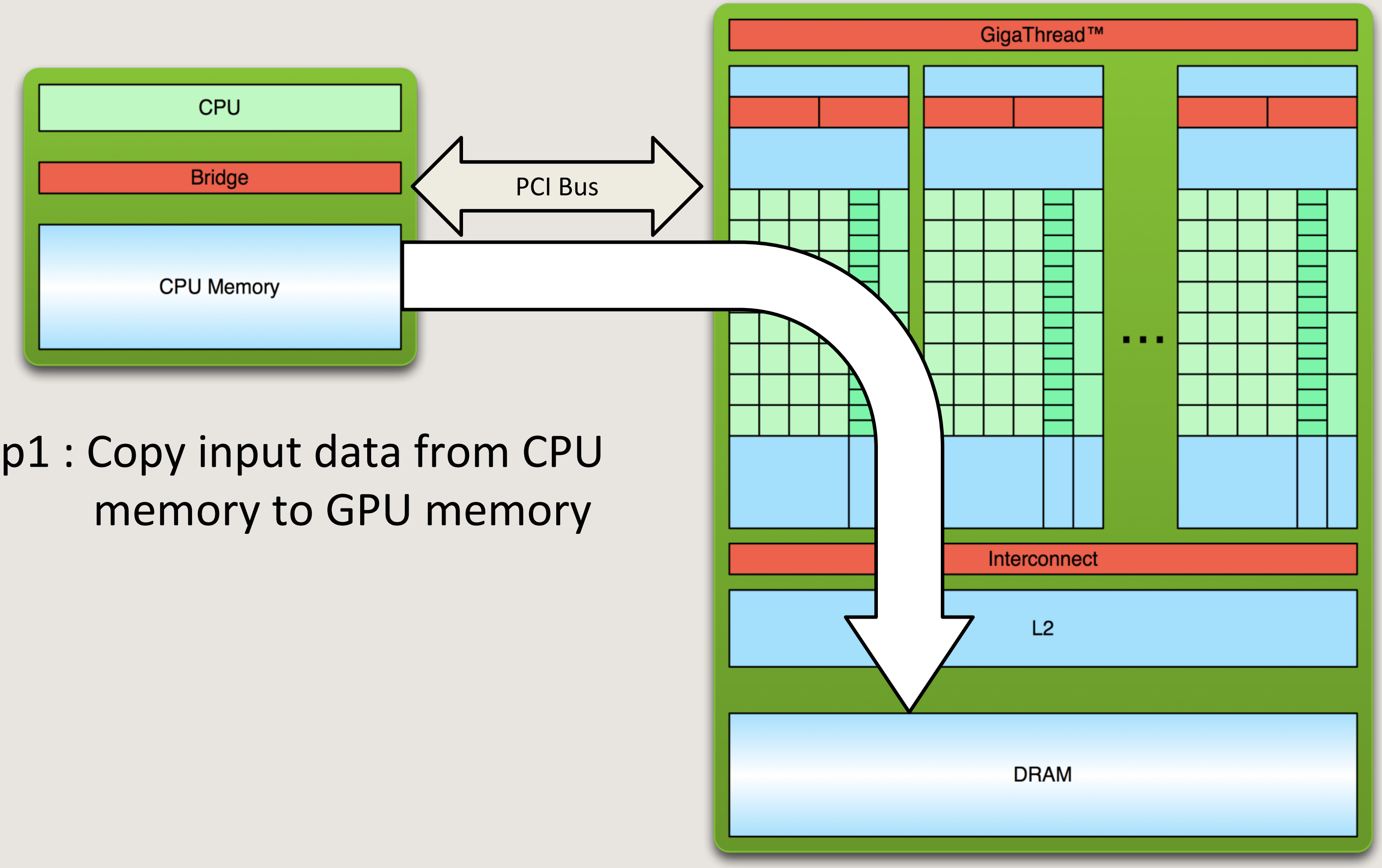
UNC CHARLOTTE

Outlines

- Background
- Motivation
- Extension to OpenMP **map** clause
- Prototype implementation for the runtime
- Benefits
- Future work
- Conclusion
- Acknowledgement

Background

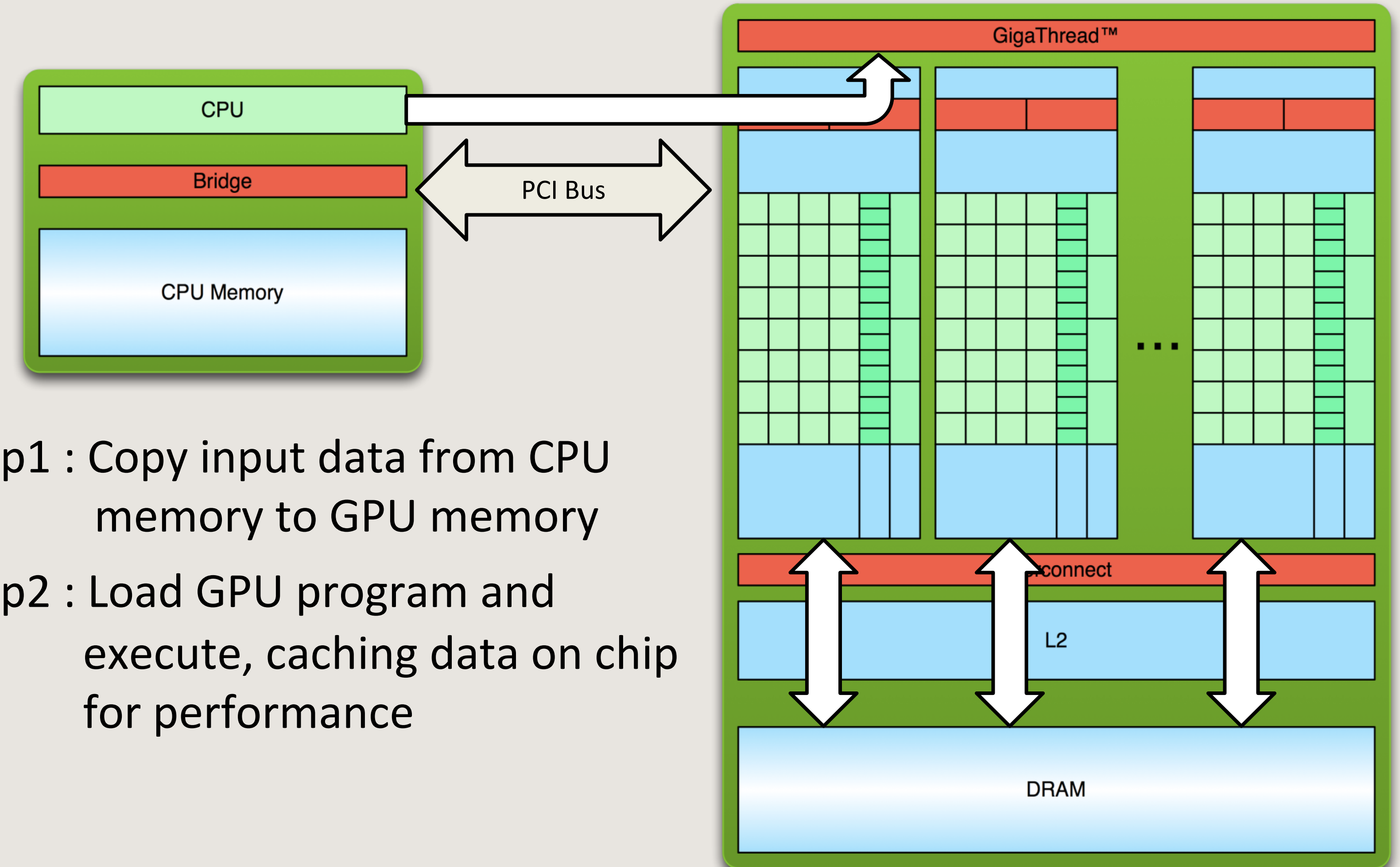
Illustrations for offloading work flow for GPU programming:



Step1 : Copy input data from CPU memory to GPU memory

Background

Illustrations for offloading work flow for GPU programming:

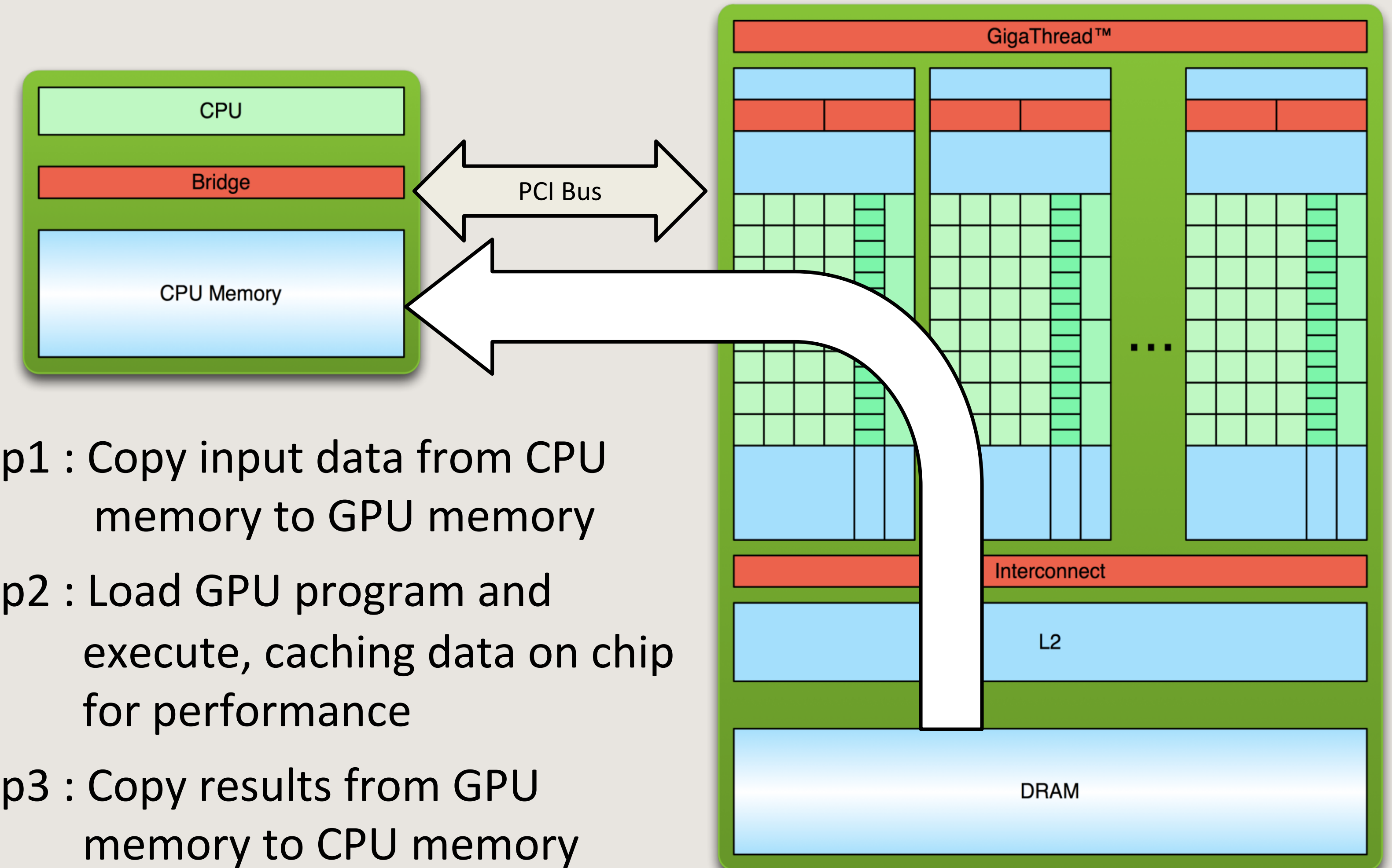


Step1 : Copy input data from CPU memory to GPU memory

Step2 : Load GPU program and execute, caching data on chip for performance

Background

Illustrations for offloading work flow for GPU programming:



Step1 : Copy input data from CPU memory to GPU memory

Step2 : Load GPU program and execute, caching data on chip for performance

Step3 : Copy results from GPU memory to CPU memory

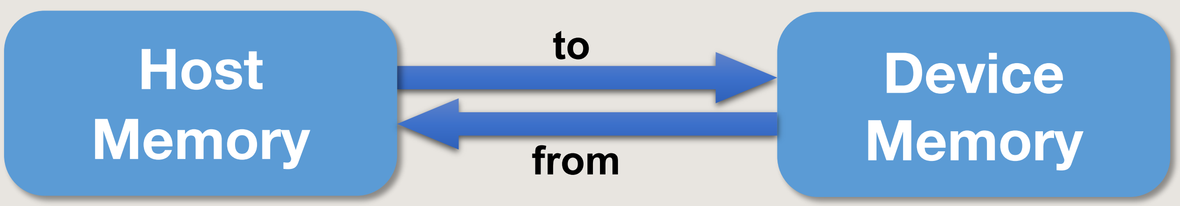
Background

A brief example for OpenMP **map** clause:

```
#pragma omp target \
    map(to:A[0:numElements],B[0:numElements]) \
    map(from:C[0:numElements])
```

map-types:

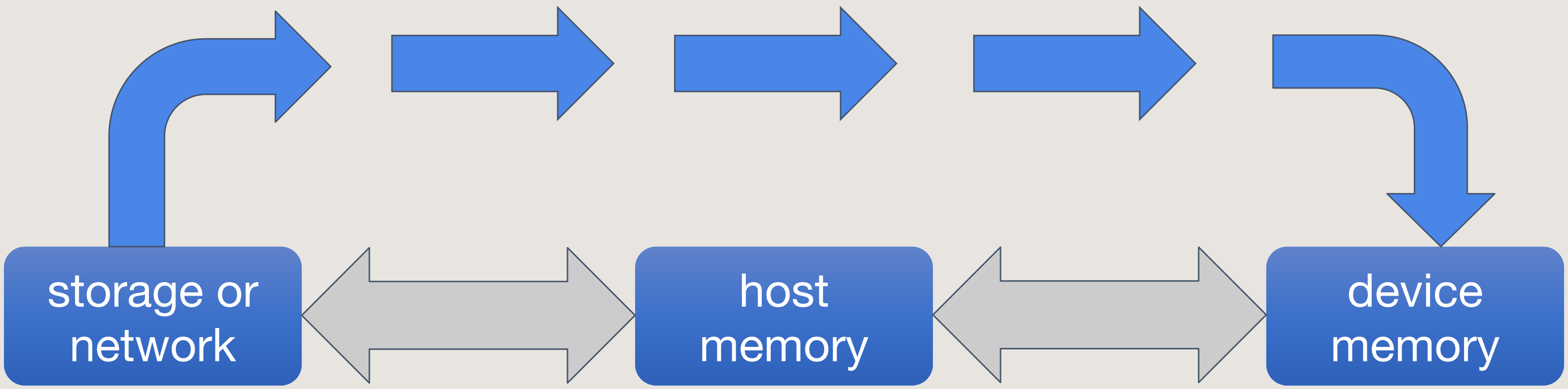
- ❑ **to** : copy A and B from host to device
- ❑ **from** : copy computing result from device to host



Motivation

Expand **map** clause to enable data copy from storage to device

- ❑Bridge storage and device memory
- ❑Reduce programming effort
- ❑Handle complex data type



***in terms of programming**

Extension to OpenMP **map** clause

First glance of the extended **map** clause:

```
#pragma omp target \
    map(to:A[0:numElements]={“data/vectorA.data”}, \
           B[0:numElements]={“data/vectorB.data”}) \
    map(from:C[0:numElements]={“data/vectorC.data”})
```

Compared with **map** clause:

- ❑ **A and B** : file for reading from storage
- ❑ **C** : file for writing back to storage

Extension to OpenMP **map** clause

An <u>optional</u> field for list item of locator-list:

list-item [= {[*data-format-driver*:] *data-location*[, *place- modifier*][, \

metadata([*place-modifier*,] *meta-identifier*)]}]

- ❑**data-format-driver** : posix, jpeg, png, …
- ❑**data-location** : local file, storage device, URL
- ❑**place-modifier** : host, hostonly
- ❑**meta-identifier** : meta_in, meta_out

Extension to OpenMP **map** clause

Extended OpenMP **map** clause example - loading image data

```
#pragma omp target \
    map(to:imgin={jpeg:“image_in.jpg”, \
    metadata(host:meta_in)}) \
    map(from:imgout={jpeg:“image_out.jpg”, \
    metadata(host:meta_out)})
```

- **list item :** imgin, imgout
- **data-format-driver :** jpeg
- **data-location :** image_in.jpg, image_out.jpg
- **place-modifier(for the metadata) :** host
- **meta-identifier :** meta_in, meta_out

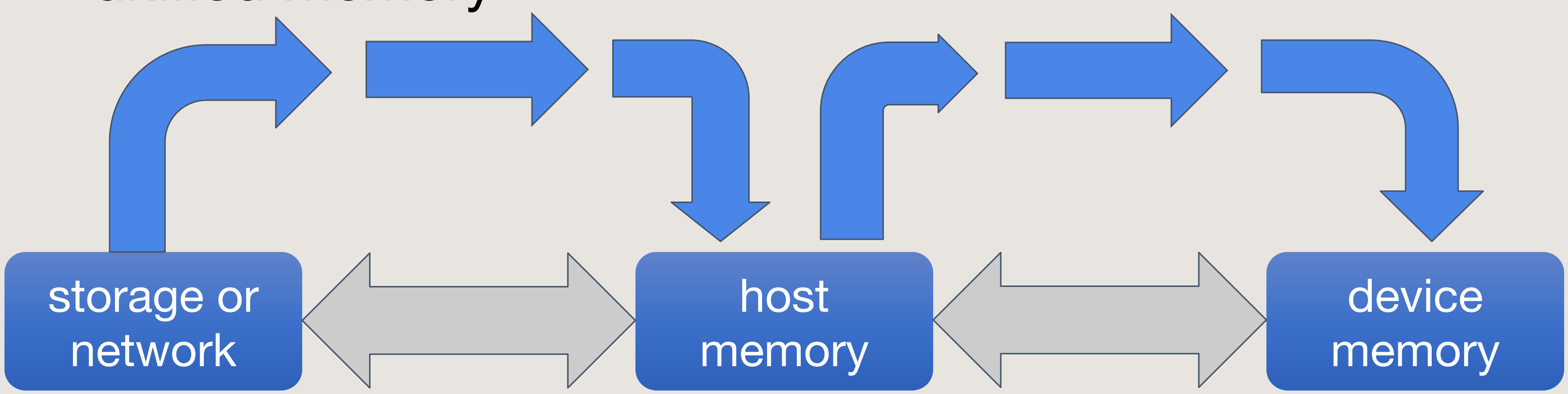
Prototype implementation for the runtime

The main idea for the implementation is that applying host memory as bounce buffer.

Two stages of data copy:

❑ storage to host: fread or imgread

❑ host to device: CUDA function calls – global memory and unified memory



***in terms of implementation**

Prototype implementation for the runtime

Example 1: POSIX stream data: Matrix Multiplication on GPU

```
fd = fopen("data/vectorA.data", "rb");
fread(tA, sizeof(float), N*K, fd);
fclose(fd);
...
cudaMalloc(&A, sizeof(float)*N*K);
cudaMemcpy(A, tA, sizeof(float)*N*K,
cudaMemcpyHostToDevice);
...
cudaMalloc(&C, sizeof(float)*N*M);
 ...
float *h_C = (float*)malloc(sizeof(float)*N*M);
...
// MM kernel
...
cudaMemcpy(h_C, C, sizeof(float)*N*M,
cudaMemcpyDeviceToHost);
...
FILE *f3;
f3 = fopen("data/vectorC.data", "wb");
fwrite(h_C, sizeof(float), N*M, f3);
fclose(f3);
```

map (to:A[0:numElements] ={“data/vectorA.data”}

host memory is used as bulk bounce buffer

map(from:C[0:numElements]={“data/vectorC.data”}

Prototype implementation for the runtime

Example 2: POSIX stream data: Matrix Multiplication on GPU

```
fd = fopen(“data/vectorA.data”, "rb");
cudaMallocManaged(&A, sizeof(float)*N*K);
fread(A, sizeof(float), N*K, fd);
fclose(fd);
...
cudaMallocManaged(&C, sizeof(float)*N*M);
...
// MM kernel
...
FILE *f3;
f3 = fopen("data/vectorC.data", "wb");
fwrite(C, sizeof(float), N*M, f3);
fclose(f3);
```

map(to:A[0:numElements]={“data/vectorA.data”}

host memory is used as page bounce buffer

map(from:C[0:numElements]={“data/vectorC.data”}

Prototype implementation for the runtime

Example 3: image data: Image Smoothing on GPU

```
uchar* imgin_d, imgout_d, imgout_h;
uchar* gpu_filter(uchar*);
Mat image = cv::imread("image_in.jpg");
size_t img_size = input.ncols * input.nrows;
cudaMalloc(imgin_d, img_size);
cudaMalloc(imgout_d, img_size);
malloc(imgout, img_size);
// copy data HtoD
cudaMemcpy(imgin_d, image.data, img_size,
cudaMemcpyHostToDevice);
// run GPU kernel
imgout_d = gpu_filter(imgin_d);
// copy data DtoH
cudamemcpy(imgout_h, imgout_d, img_size,
cudaMemcpyDeviceToHost);
// write result to a new file
image.data = imgout_h;
cv::imwrite("image_out.jpg", image);
```

map(to: imgin={jpeg: ”image_in.jpg”, metadata(host: meta_in)})

host memory is used as bulk bounce buffer

imgout={“image_out.jpg”, metadata(host: meta_out)})

Prototype implementation for the runtime

Example 4: image data: Image Smoothing on GPU

```
uchar* imgin, imgout;
uchar* gpu_filter(uchar*);
Mat image = cv::imread("image_in.jpg");
size_t img_size = input.ncols * input.nrows;
cudaMallocManaged(imgin, img_size);
cudaMallocManaged(imgout, img_size);
memcpy(imgin, image.data, img_size);
// run GPU kernel
imgout = gpu_filter(imgin);
// write result to a new file
image.data = imgout;
cv::imwrite("image_out.jpg", image);
```

map(to: imgin={jpeg: ”image_in.jpg”, metadata(host: meta_in)})

host memory is used as page bounce buffer

imgout={jpeg: “image_out.jpg”, metadata(host: meta_out)})

NOTE:memcpy is still needed here since the data to be processed is preloaded to the host memory.

Benefits

image data : smoothing, optimize paging to obtain higher performance for writing data back to storage.

Image Size	Input	Output	HtoD	DtoH	Kernel
512x512	3	274	0.066	0.061	0.204
512x1024	6	629	0.142	0.123	0.526
1024x1024	10	1285	0.338	0.706	0.853
1024x2048	20	2622	0.694	2.197	2.186
2048x2048	35	4833	1.471	5.289	3.545

TABLE I: Breakdown of execution time for image smoothing using global memory (ms)

DtoH ranges from about 30% to 150% of the kernel execution time.

Image Size	Input	Output	HtoD	DtoH	Page Fault	Kernel
512x512	3	316	0.250	0.174	2.294	2.499
512x1024	6	660	0.303	0.239	3.096	3.176
1024x1024	10	1288	0.381	0.305	2.718	3.491
1024x2048	19	2637	0.813	0.600	5.314	7.241
2048x2048	37	4823	1.381	1.085	8.693	11.785

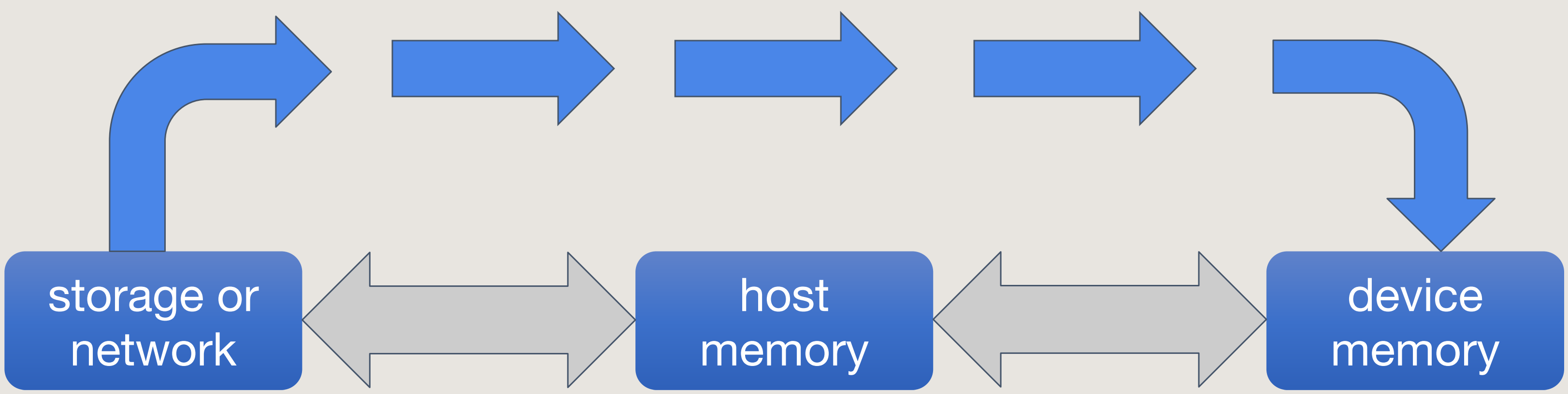
TABLE II: Breakdown of execution time for image smoothing using unified memory (ms)

Page Fault ranges from about 73% to 96% of the kernel execution time.

Future work

Optimization on data copy between storage and device:

- ❑ NVIDIA GPUDirect Storage - no bounce buffer at all
- ❑ cudaHostRegister with mmap - host to device, pinned memory applied
- ❑ Linux Direct Access(DAX) - storage to host, involving NVDIMM



Conclusion

- ❑ Offloading work flow shows that there should be less effort on loading data from storage to device in terms of programming
- ❑ Add optional elements to OpenMP **map** clause to get access to data copy from storage to device for users
- ❑ Two implementation ideas to use host memory as bulk bounce buffer and page bounce buffer
- ❑ Performance results show potential for further optimization

Acknowledgement

- ❑ This material is based upon work supported by the National Science Foundation under Grant No. 1833332 and 1652732.
- ❑ MCHPC'19 Committee

SC19

Denver, CO | hpc is now.

Thank you!

Questions?