CUDA Asynchronous Memory Usage and Execution

Yukai Hung
a0934147@gmail.com
Department of Mathematics
National Taiwan University
Page-Locked Memory
Page-Locked Memory

- Regular pageable and page-locked or pinned host memory
  - use too much page-locked memory reduces system performance

physical memory

restore paging memory into physical memory

local disk for virtual memory
Page-Locked Memory

- Regular pageable and page-locked or pinned host memory
- Copy between page-locked memory and device memory can be performed concurrently with kernel execution for some devices

load blue block to shared memory
compute blue block on shared memory
and load next block to shared memory
Regular pageable and page-locked or pinned host memory
- copy between page-locked memory and device memory can be performed concurrently with kernel execution for some devices
Page-Locked Memory

- Regular pageable and page-locked or pinned host memory
  - use page-locked host memory can support executing more than one device kernel concurrently for compute capability 2.0 hardware
Page-Locked Memory

- Portable memory
  - the block of page-locked memory is only available for the thread that allocates it by the default setting, use portable memory flag to share the page-locked memory with other threads
Page-Locked Memory

How to allocate portable memory?

```c
float* pointer;

//allocate host page-locked write-combining memory
cudaHostAlloc((void**)&pointer, bytes, cudaHostAllocPortable);

//free allocated memory space
cudaFreeHost(pointer);
```
Page-Locked Memory

- **Write-Combining memory**
  - page-locked memory is allocated as cacheable by default
  - page-locked memory can be allocated as write-combining memory by using special flag, which frees up L1 and L2 cache resource usage

- **Advantage and disadvantage**
  - write-combining memory is not snooped during transfers across bus, which can improve transfer performance by up to 40%
  - reading from write-combining memory from host is slow, which should in general be used for memory that the host only write to
Page-Locked Memory

How to allocate write-combining memory?

```c
float* pointer;

//allocate host page-locked write-combining memory
cudaHostAlloc((void**) &pointer, bytes, cudaMemcpyHostAllocWriteCombined);

//free allocated memory space
cudaFreeHost(pointer);
```

<table>
<thead>
<tr>
<th>1 GB data size</th>
<th>normal</th>
<th>write-combining</th>
</tr>
</thead>
<tbody>
<tr>
<td>host to device</td>
<td>0.533522</td>
<td>0.338092</td>
</tr>
<tr>
<td>device to host</td>
<td>0.591750</td>
<td>0.320989</td>
</tr>
</tbody>
</table>
Page-Locked Memory

- Mapped memory
  - the page-locked host memory can be mapped into the address space of the device by passing special flag to allocate memory
Page-Locked Memory

- Mapped memory
  - the page-locked host memory can be mapped into the address space of the device by passing special flag to allocate memory

map host memory into device memory

host memory

host memory pointer

device memory

device memory pointer

implicit asynchronous transfer

read or write data will acts as only one memory space
Page-Locked Memory

How to allocate mapped memory?

```c
float* pointer;

//allocate host page-locked write-combining memory
cudaHostAlloc((void**)&pointer, bytes, cudaHostAllocMapped);

//free allocated memory space
cudaFreeHost(pointer);
```
Page-Locked Memory

◉ Check the hardware is support or not?
  - check the hardware properties to ensure it is available for mapping host page-locked memory with device memory

```c
cudaDeviceProp deviceprop;

//query the device hardwared properties
//the structure records all device properties
cudaGetDeviceProperties(&deviceprop, 0);

//check the map memory is available or not
if (!deviceprop.canMapHostMemory)
  printf("cudaError:cannot map host to devicememory\n");
```
What is the property structure contents?
#define size 1048576

int main(int argc, char** argv)
{
    int loop;

    float residual;

    float *h_a, *d_a;
    float *h_b, *d_b;
    float *h_c, *d_c;

    cudaDeviceProp deviceprop;

    //query the device hardwared properties
    //the structure records all device properties
cudaGetDeviceProperties(&deviceprop, 0);

    //check the map memory is available or not
    if(!deviceprop.canMapHostMemory)
        printf("cudaError: cannot map host to device memory\n");
Mapped Memory

//this flag must be set in order to allocate pinned
//host memory that is accessible to the device
cudaSetDeviceFlags(cudaDeviceMapHost);

//allocate host page-locked and accessible to the device memory
//maps the memory allocation on host into cuda device address
cudaHostAlloc((void**)&h_a,sizeof(float)*size,cudaHostAllocMapped);
cudaHostAlloc((void**)&h_b,sizeof(float)*size,cudaHostAllocMapped);
cudaHostAlloc((void**)&h_c,sizeof(float)*size,cudaHostAllocMapped);

//initialize host vectors
for(loop=0;loop<size;loop++)
{
    h_a[loop]=(float)rand()/(RAND_MAX-1);
    h_b[loop]=(float)rand()/(RAND_MAX-1);
}

//pass back the device pointer and map with host
cudaHostGetDevicePointer((void**)&d_a,(void*)h_a,0);
cudaHostGetDevicePointer((void**)&d_b,(void*)h_b,0);
cudaHostGetDevicePointer((void**)&d_c,(void*)h_c,0);
//execute device kernel for vector addition
vectorAdd<<<(int)ceil((float)size/256),256s>>>(d_a,d_b,d_c,size);
cudaThreadSynchronize();

//check the result residual value
for(loop=0,residual=0.0;loop<size;loop++)
residual=residual+(h_a[loop]+h_b[loop]-h_c[loop]);

printf("residual value is %f\n",residual);

//free the memory space which must have been returned
//by a previous call to cudaMallocHost or cudaHostAlloc
cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);

//catch and check cuda error message
if((error=cudaGetLastError())!=cudaSuccess)
printf("cudaError:%s\n",cudaGetErrorString(error));

return 0;
}
__global__ void vectorAdd(float* da, float* db, float* dc, int size)
{
    int index;

    // calculate each thread global index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < size)
    // each thread compute one component
    dc[index] = da[index] + db[index];

    return;
}
Several advantages

- there is no need to allocate a block in device memory and copy data between this block and block in host memory, the data transfers are implicitly performed as needed by the kernel

- there is no need to use streams to overlap data transfers with kernel execution, the kernel-originated data transfers overlap with kernel execution automatically

- mapped memory is able to exploit the full duplex of the PCI express bus by reading and writing at the same time, since memory copy only move data in one direction, half duplex
Page-Locked Memory

- Several disadvantages

  - the page-locked memory is shared with host and device, any application must avoid write on the both side simultaneously

  - the atomic functions operating on mapped page-locked memory are not atomic from the point of view of the host or other devices
Portable and mapped memory
- the page-locked host memory can be allocated as both portable and mapped memory, such that each host thread can map the same page-locked memory into different device address.
Page-Locked Memory

- Integrated system
  - the mapped page-locked memory is very suitable on integrated system that utilize the a part of host memory as device memory
  - check the integrated field on cuda device properties structure
  - mapped memory is faster, if data only read from or write to global memory once, the coalescing is even more important with mapped memory in order to reduce the data transfer times
Asynchronous Execution
Asynchronous Execution

- Asynchronous execution
  - some functions are supported asynchronous launching in order to facilitate concurrent execution between host and device resource
  - control is returned to the host thread before the work is finished

- transfer data between host and device
- perform some device kernels
- perform some host functions

overlapping
Asynchronous Execution

- Asynchronous execution
  - some functions are supported asynchronous launching in order to facilitate concurrent execution between host and device resource
  - control is returned to the host thread before the work is finished

perform device kernel launch
kernel<<<blocknum,blocksize,0,stream>>>(...)

perform data transfer between host and device
perform data transfer between device and device
cudaMemcpyAsync(destination, source, bytes, direction, stream);

perform global memory set
### Asynchronous Execution

```c
#define size 1048576

int main(int argc, char** argv)
{
    int loop;
    int bytes;

    float *h_a, *d_a;
    float *h_b, *d_b;
    float *h_c, *d_c;

    // allocate host page-locked memory
    cudaMallocHost((void**)&h_a, sizeof(float)*size);
    cudaMallocHost((void**)&h_b, sizeof(float)*size);
    cudaMallocHost((void**)&h_c, sizeof(float)*size);

    // allocate device global memory
    cudaMalloc((void**)&d_a, sizeof(float)*size);
    cudaMalloc((void**)&d_b, sizeof(float)*size);
    cudaMalloc((void**)&d_c, sizeof(float)*size);
```
Asynchronous Execution

cudaEvent_t stop;
cudaEvent_t start;

//create an event object which is used to
//record device execution elapsed time
cudaCreateEvent(&stop);
cudaCreateEvent(&start);

//initialize host vectors
for(loop=0;loop<size;loop)
{
    h_a[loop]=(float)rand()/(RAND_MAX-1);
    h_b[loop]=(float)rand()/(RAND_MAX-1);
}
Asynchronous Execution

bytes = sizeof(float) * size;

// set time event recorder
cudaEventRecord(start, 0);

// copy data from host to device memory asynchronously
cudaMemcpyAsync(d_a, h_a, bytes, cudaMemcpyHostToDevice, 0);
cudaMemcpyAsync(d_b, h_b, bytes, cudaMemcpyHostToDevice, 0);

// execute device kernel asynchronously
vectorAdd<<<(int)ceil((float)size/256, 256, 0, 0)>>>(d_a, d_, d_c, size);

// copy data from device to host memory asynchronously
cudaMemcpyAsync(h_c, d_c, bytes, cudaMemcpyDeviceToHost, 0);

// set time event recorder
cudaEventRecord(stop, 0);
Asynchronous Execution

```c
counter=0;

// increase the counter before the queried
// cuda event has actually been finished
while(cudaEventQuery(stop)==cudaErrorNotReady)
counter=counter+1;

// calculate device execution elapsed time
cudaEventElapsedTime(&elapsed,start,stop);

// check the result residual value
for(loop=0,residual=0.0;loop<size;loop++)
residual=residual+(h_c[loop]-h_a[loop]-h_b[loop]);

printf("counter:%d\n",counter);
printf("residual:%f\n",residual);
```
Asynchronous Execution

//free the memory space which must have been returned by a previous call to cudaMemcpyHost or cudaMemcpy
cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);

//free the device memory space
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

//free the cuda event object
cudaEventDestroy(stop);
cudaEventDestroy(start);

//catch and check cuda error message
if((error=cudaGetLastError())!=cudaSuccess)
printf("cudaError:%s\n",cudaGetErrorString(error));

return 0;
}
Asynchronous Execution

```c
__global__ void vectorAdd(float* da, float* db, float* dc, int size) {
    int index;
    // calculate each thread global index
    index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        // each thread computer one component
        dc[index] = da[index] + db[index];
    }
    return;
}
```
Stream

- Stream
  - applications manage concurrency through stream
  - a stream is a sequence of commands that execute in order
  - all device requests made from the host code are put into a queues
Stream

How to create a stream?

```c
cudaStream_t stream;

//create an asynchronous new stream
cudaStreamCreate(&stream);

//destroy stream
cudaStreamDestroy(stream);
```
Stream

- different streams may execute their commands or host requests out of order with respect to one another or concurrently, but the same stream is still a sequence of commands that execute in order.
Stream

```c
#define snum 10
#define size 1048576

int main(int argc, char** argv)
{
    int loop;
    int bytes;

    float *h_a, *d_a;
    float *h_b, *d_b;
    float *h_c, *d_c;

    cudaStream_t stream[snum];

    //create new asynchronous stream
    //which acts as device work queue
    for(loop=0; loop<snum; loop++)
        cudaStreamCreate(stream+loop);
```
//allocate host page-locked memory
cudaMallocHost((void**) &h_a, sizeof(float)*size*snum);
cudaMallocHost((void**) &h_b, sizeof(float)*size*snum);
cudaMallocHost((void**) &h_c, sizeof(float)*size*snum);

//allocate device global memory
cudaMalloc((void**) &d_a, sizeof(float)*size*snum);
cudaMalloc((void**) &d_b, sizeof(float)*size*snum);
cudaMalloc((void**) &d_c, sizeof(float)*size*snum);

//initialize host vectors
for (loop=0; loop<size*snum; loop++)
{
    h_a[loop] = (float)rand()/(RAND_MAX-1);
    h_b[loop] = (float)rand()/(RAND_MAX-1);
}

Stream

Stream

// put all the works into default stream
// executes all works by using only one stream
for(loop=0; loop<snum; loop++)
{
    bytes = sizeof(float) * size;

    sp1 = h_a + loop * size; dp1 = d_a + loop * size;
    sp2 = h_b + loop * size; dp2 = d_b + loop * size;
    sp3 = d_c + loop * size; dp3 = h_c + loop * size;

    // copy data from host to device memory asynchronously
    cudaMemcpyAsync(dp1, sp1, bytes, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync(dp2, sp2, bytes, cudaMemcpyHostToDevice, 0);

    // execute device kernel asynchronously
    kernel <<<blocknum, blocksize, 0, 0>>>(d_a, d_b, d_c, size);

    // copy data from device to host memory asynchronously
    cudaMemcpyAsync(dp3, sp3, bytes, cudaMemcpyDeviceToHost, 0);
}

// wait until the stream is finished
cudaThreadSynchronize();
//put all the works into different asynchronous streams
//each stream only executes three copies and one kernel
for(loop=0;loop<snum;loop++)
{
    bytes=sizeof(float)*size;
    sp1=h_a+loop*size; dp1=d_a+loop*size;
    sp2=h_b+loop*size; dp2=d_b+loop*size;
    sp3=d_c+loop*size; dp3=h_c+loop*size;

    //copy data from host to device memory asynchronously
    cudaMemcpyAsync(dp1,sp1,bytes,cudaMemcpyHostToDevice,
                    stream[loop]);
    cudaMemcpyAsync(dp2,sp2,bytes,cudaMemcpyHostToDevice,
                    stream[loop]);

    //execute device kernel asynchronously
    kernel<<<blocknum,blocksize,0,stream[loop]>>>(d_a,d_b,d_c,size);

    //copy data from device to host memory asynchronously
    cudaMemcpyAsync(dp3,sp3,bytes,cudaMemcpyDeviceToHost,
                    stream[loop]);
}

//wait until all stream are finished
cudaThreadSynchronize();
//free the memory space which must have been returned
d by a previous call to cudaMallocHost or cudaHostAlloc

cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);

//free the device memory space
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

//free the asynchronous streams
for(loop=0;loop<snum;loop++)
    cudaStreamDestroy(stream[loop]);

return 0;
}
__global__ void vectorAdd(float* da, float* db, float* dc, int size) {
    int loop;
    int index;

    volatile float temp1;
    volatile float temp2;

    // calculate each thread global index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < size)
        for (loop = 0; loop < iteration; loop++)
            {
            temp1 = da[index];
            temp2 = db[index];
            dc[index] = temp1 + temp2;
            }

    return;
}
Stream

- How about the performance?

<table>
<thead>
<tr>
<th></th>
<th>Fermi C2050</th>
<th>Tesla C1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>single stream</td>
<td>64.096382</td>
<td>180.179825</td>
</tr>
<tr>
<td>multiple stream</td>
<td>31.996338</td>
<td>166.010757</td>
</tr>
</tbody>
</table>
Stream controlling

`cudaThreadSynchronize()`
called in the end to make sure all streams are finished before preceding further, it forces the runtime to wait until all device tasks or commands in all asynchronous streams have completed

`cudaStreamSynchronize()`
force the runtime to wait until all preceding device tasks or host commands in one specific stream have completed

`cudaStreamQuery()`
provide applications with a way to know if all preceding device tasks or host commands in a stream have completed
Stream controlling

`cudaStreamDestroy()`
wait for all preceding tasks in the give stream to complete before destroying the stream and returning control to the host thread, which is blocked until the stream finished all commands or tasks
Overlap of data transfer and kernel execution transfer data between host page-locked memory and device memory and kernel execution can be performed concurrently.
Overlap of data transfer and kernel execution
transfer data between host page-locked memory and device memory and kernel execution can be performed concurrently
any application may query the hardware capability by calling the device manage function and checking the property flag

```c
    cudaDeviceProp deviceprop;
    //query the device hardware properties
    //the structure records all device properties
    cudaGetDeviceProperties(&deviceprop, 0);

    //check the overlapping is available or not
    if(!deviceprop.deviceOverlap)
        printf("cudaError: cannot overlap kernel and transfer\n");
```
- Concurrent kernel execution
  some hardware can execute multiple kernels concurrently
Concurrent kernel execution

some hardware can execute multiple kernels concurrently

any application may query the hardware capability by calling the device manage function and checking the property flag

```c
cudaDeviceProp deviceprop;

//query the device hardware properties
//the structure records all device properties
cudaGetDeviceProperties(&deviceprop, 0);

//check the concurrent kernels is available or not
if (!deviceprop.concurrentKernels)
    printf("cudaError: cannot use concurrent kernels\n");
```
Stream

- Concurrent kernel execution
  some hardware can execute multiple kernels concurrently
  any application may query the hardware capability by calling
  the device manage function and checking the property flag
  the kernels that may use many textures or registers or shared
  memory are less likely to execute with other kernels concurrently
Concurrent data transfer

Some devices can perform copy data from page-locked memory to device memory with copy data from device memory to host page-locked memory concurrently.

Copy data from device to page-locked memory

Copy data from page-locked to device memory

Host thread

Device driver

NEW device

OLD device
Streams on the matrix-matrix multiplication
Event

The runtime provides a way to closely monitor the device progress by letting the program record events at any point in the program.

```c
cudaEvent_t event1;
cudaEvent_t event2;

//create and initialize event
cudaEventCreate(&event1);
cudaEventCreate(&event2);

//insert event recorder into stream
cudaEventRecord(event1, stream);
cudaEventRecord(event2, stream);

//destroy created event recorder
cudaEventDestroy(event1);
cudaEventDestroy(event2);
```
Event

Event

```
cudaEventSynchronize()
cudaEventQuery()
```

this function blocks until the event has actually been recorded, since the event recorder is an asynchronous method

provide any applications with a way to know if one specific event recorder in the stream have completed, which returns cudaSuccess
Multi-GPU
Multi-GPU

- GPU can not share global memory
- one GPU can not access another GPUs memory directly
- application code is responsible for moving data between GPUs
A host thread can maintain one context at a time
- need as many host threads as GPUs to maintain all device
- multiple host threads can establish context with the same GPU
hardware diver handles time-sharing and resource partitioning
Device management calls

`cudaGetDeviceCount()`
returns the number of devices on the current system with compute capability greater or equal to 1.0, that are available for execution

`cudaSetDevice()`
set the specific device on which the active host thread executes the device code. If the host thread has already initialized he cuda runtime by calling non-device management runtime functions, returns error

must be called prior to context creation, fails if the context has already been established, one can forces the context creation with cudaFree(0)

`cudaGetDevice()`
returns the device on which the active host thread executes the code
Device management calls

```
cudaThreadExit()
```

explicitly clean up all runtime-related resource associated with the calling host thread, any subsequent calls reinitializes the runtime, this function is implicitly called on the host thread exit

```
cudaGetDeviceProperties()
```

returns the properties of one specific device, this is useful when a system contains different devices in order to choose best devices
Reference

- Mark Harris  http://www.markmark.net/
- Wei-Chao Chen  http://www.cs.unc.edu/~ciao/
- Wen-Mei Hwu  http://impact.crhc.illinois.edu/people/current/hwu.php