Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

MCS 572 Lecture 39
Introduction to Supercomputing
Jan Verschelde, 17 April 2023

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

page locked or pinned memory

In contrast to regular pageable host memory, the runtime provides functions to allocate (and free) *page locked* memory.

Another name for memory that is page locked is *pinned*.

Using page locked memory has several benefits:

- Copies between page locked memory and device memory can be performed concurrently with kernel execution.
- Page locked host memory can be mapped into the address space of the device, eliminating the need to copy, we say *zero copy*.
- Bandwidth between page locked host memory and device may be higher.

Page locked host memory is a scarce resource.

The NVIDIA CUDA Best Practices Guide assigns a low priority to zero-copy operations (i.e.: mapping host memory to the device).

the output of `bandwidthTest` on V100

In `/usr/local/cuda/samples/1_Utilities/bandwidthTest`:

```
$ ./bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Quadro GV100
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes) Bandwidth(GB/s)
   32000000 12.3

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes) Bandwidth(GB/s)
   32000000 13.2

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes) Bandwidth(GB/s)
   32000000 541.1

Result = PASS
```

allocating and mapping pinned host memory

To allocate page locked memory, we use `cudaHostAlloc()` and to free the memory, we call `cudaFreeHost()`.

To map host memory on the device:

- The flag `cudaHostAllocMapped` must be given to `cudaHostAlloc()` when allocating host memory.
- A call to `cudaHostGetDevicePointer()` maps the host memory to the device.

If all goes well, then no copies from host to device memory and from device to host memory are needed.

Not all devices support pinned memory, it is recommended practice to check the device properties (see the `deviceQuery` in the SDK).

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

using pinned memory

```
Quadro GV100 supports mapping host memory.

The error code of cudeHostAlloc : 0

Squaring 32 numbers 1 2 3 4 5 6 7 8 9 10 11 12 13 \
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32...


The fail code of cudaHostGetDevicePointer : 0

After squaring 32 numbers 1 4 9 16 25 36 49 64 81 \
100 121 144 169 196 225 256 289 324 361 400 441 484 \
529 576 625 676 729 784 841 900 961 1024...

$
```

the program pinnedmemoryuse.cu

```
/* This program illustrates pinned (page locked) memory. */

#include <stdio.h>

int checkDeviceProp ( cudaDeviceProp p );
/*
 * Returns 0 if the device does not support mapping
 * host memory, returns 1 otherwise. */
```

checking the device

```
int checkDeviceProp ( cudaDeviceProp p )
{
   int support = p.canMapHostMemory;

   if(support == 0)
      printf("%s does not support mapping host memory.\n",
             p.name);
   else
      printf("%s supports mapping host memory.\n",p.name);

   return support;
}
```

a simple kernel

```
__global__ void Square ( float *x )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in x[i]. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   x[i] = x[i]*x[i];
}
```

the main program

```
void square_with_pinned_memory ( int n );
/*
 * Illustrates the use of pinned memory to square
 * a sequence of n numbers. */

int main ( int argc, char* argv[] )
{
   cudaDeviceProp dev;

   cudaGetDeviceProperties(&dev,0);

   int success = checkDeviceProp(dev);

   if(success != 0)
      square_with_pinned_memory(32);

   return 0;
}
```

allocating pinned memory

```
void square_with_pinned_memory ( int n )
{
   float *xhost;
   size_t sz = n*sizeof(float);
   int error = cudaHostAlloc((void**)&xhost,
                  sz,cudaHostAllocMapped);
   printf("\nThe error code of cudeHostAlloc : %d\n",
          error);

   for(int i=0; i<n; i++) xhost[i] = (float) (i+1);
   printf("\nSquaring %d numbers",n);
   for(int i=0; i<n; i++) printf(" %d",(int) xhost[i]);
   printf("...\n\n");
```

mapping host memory

```
   float *xdevice;

   int fail = cudaHostGetDevicePointer
                  ((void**)&xdevice,(void*)xhost,0);
   printf("\nThe fail code of cudaHostGetDevicePointer : \
          %d\n",fail);

   Square<<<1,n>>>(xdevice);

   cudaDeviceSynchronize();

   printf("\nAfter squaring %d numbers",n);
   for(int i=0; i<n; i++) printf(" %d",(int) xhost[i]);
   printf("...\n\n");

   cudaFreeHost(xhost);
}
```

applications of pinned memory

A data transfer between host and device with `cudaMemcpy()` is *blocking*: control to the host is returned only after transfer is complete.

It is possible to overlap data transfers with computation using `cudaMemcpyAsync()`, which is not blocking as it returns control immediately to the host.

Asynchronous data transfers require pinned memory.

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

streams and concurrency

The Fermi architecture supports the simultaneous execution of kernels.

Benefits:

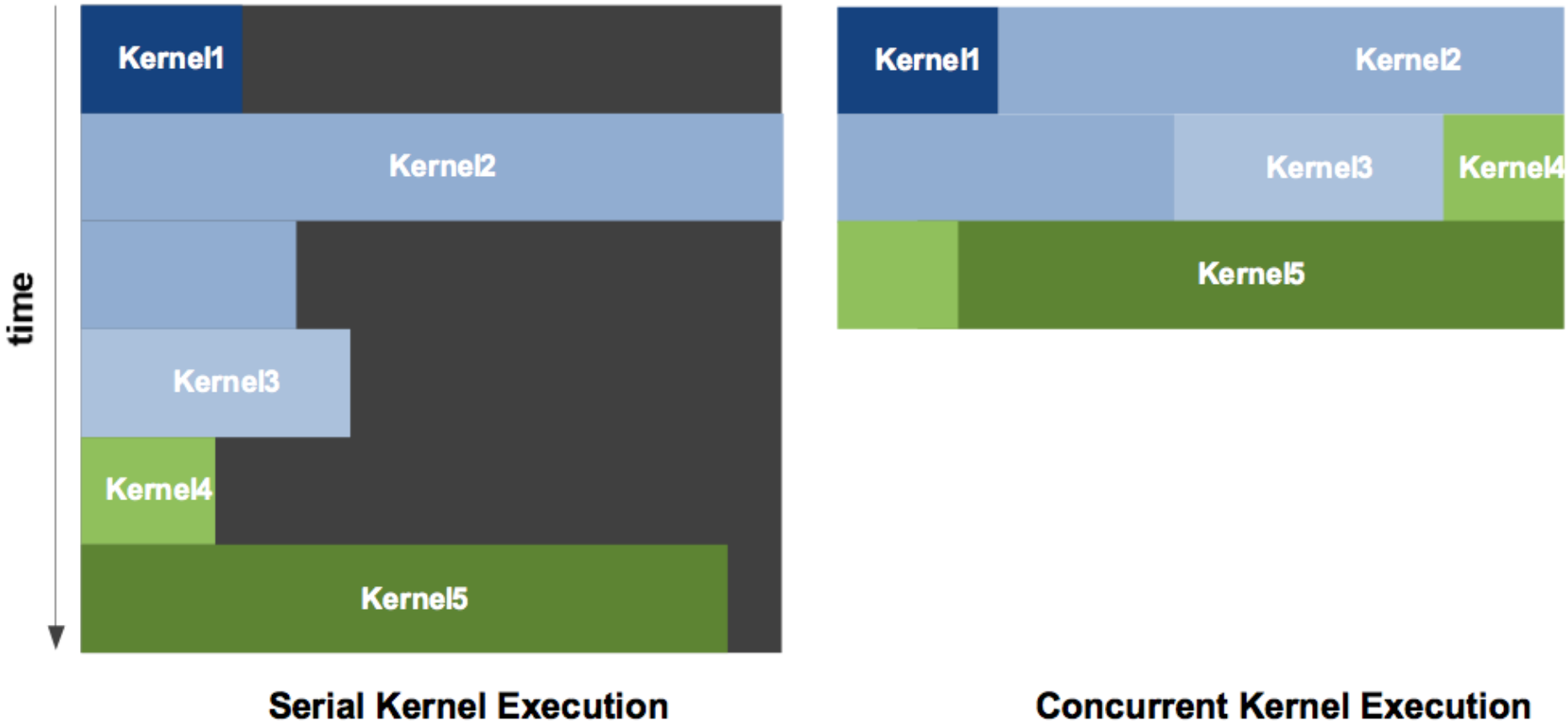
- Simultaneous execution of small kernels utilize whole GPU.
- Overlapping kernel execution with device to host memory copy.

A *stream* is a sequence of commands that execute in order.

Different streams may execute concurrently.

The maximum number of kernel launches that a device can execute concurrently is four on the K20C.

concurrent kernel execution



from the NVIDIA Fermi Compute Architecture Whitepaper

concurrent copy and kernel execution (4 streams)

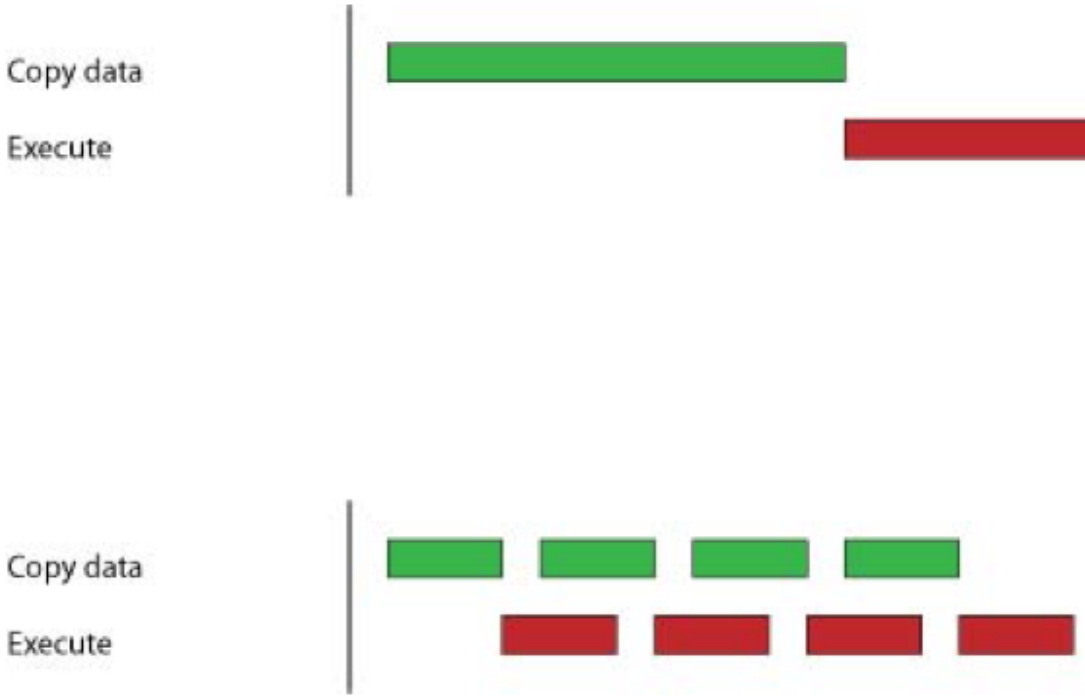


Figure 3.1 Timeline Comparison for Sequential (top) and Concurrent (bottom) Copy and Kernel Execution

from the NVIDIA CUDA Best Practices Guide

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

squaring numbers

```
$ /tmp/concurrent
Quadro GV100 supports concurrent kernels
  compute capability : 7.0
  number of multiprocessors : 80

Launching 4 kernels on 16 numbers 1 2 3 4 5 6 7 8 9 10 \
11 12 13 14 15 16...

the 16 squared numbers are 1 4 9 16 25 36 49 64 81 100 \
121 144 169 196 225 256
$
```

a simple kernel

```
__global__ void Square ( float *x, float *y )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in y[i]. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   y[i] = x[i]*x[i];
}
```

checking if concurrency is supported

```
int checkDeviceProp ( cudaDeviceProp p )
{
   int support = p.concurrentKernels;

   if(support == 0)
      printf("%s does not support concurrent kernels\n",
             p.name);
   else
      printf("%s supports concurrent kernels\n",p.name);

   printf("  compute capability : %d.%d \n",
          p.major,p.minor);
   printf("  number of multiprocessors : %d \n",
          p.multiProcessorCount);

   return support;
}
```

the main program

```
void launchKernels ( void );
/*
 * Launches concurrent kernels on arrays of floats. */

int main ( int argc, char* argv[] )
{
   cudaDeviceProp dev;
   cudaGetDeviceProperties(&dev,0);

   int success = checkDeviceProp(dev);
   if(success != 0) launchKernels();

   return 0;
}
```

allocating memories

```
void launchKernels ( void )
{
   const int nbstreams = 4;
   const int chunk = 4;
   const int nbdata = chunk*nbstreams;

   float *xhost;
   size_t sz = nbdata*sizeof(float);

   cudaMallocHost((void**)&xhost,sz);  // PINNED MEMORY

   for(int i=0; i<nbdata; i++) xhost[i] = (float) (i+1);
   printf("\nLaunching %d kernels on %d numbers",
          nbstreams,nbdata);
   for(int i=0; i<nbdata; i++)
      printf(" %d",(int) xhost[i]);
   printf("...\n\n");
   float *xdevice; cudaMalloc((void**)&xdevice,sz);
   float *ydevice; cudaMalloc((void**)&ydevice,sz);
```

asynchronous concurrent execution

```
cudaStream_t s[nbstreams];
for(int i=0; i<nbstreams; i++) cudaStreamCreate(&s[i]);

for(int i=0; i<nbstreams; i++)
   cudaMemcpyAsync
      (&xdevice[i*chunk],&xhost[i*chunk],
       sz/nbstreams,cudaMemcpyHostToDevice,s[i]);

for(int i=0; i<nbstreams; i++)
   Square<<<1,chunk,0,s[i]>>>
      (&xdevice[i*chunk],&ydevice[i*chunk]);
```

synchronization

```
   for(int i=0; i<nbstreams; i++)
      cudaMemcpyAsync
         (&xhost[i*chunk],&ydevice[i*chunk],
          sz/nbstreams,cudaMemcpyDeviceToHost,s[i]);

   cudaDeviceSynchronize();

   printf("the %d squared numbers are",nbdata);
   for(int i=0; i<nbdata; i++)
      printf(" %d",(int) xhost[i]);
   printf("\n");

   for(int i=0; i<nbstreams; i++) cudaStreamDestroy(s[i]);
   cudaFreeHost(xhost);
   cudaFree(xdevice); cudaFree(ydevice);
}
```

Volta Multi-Process Service

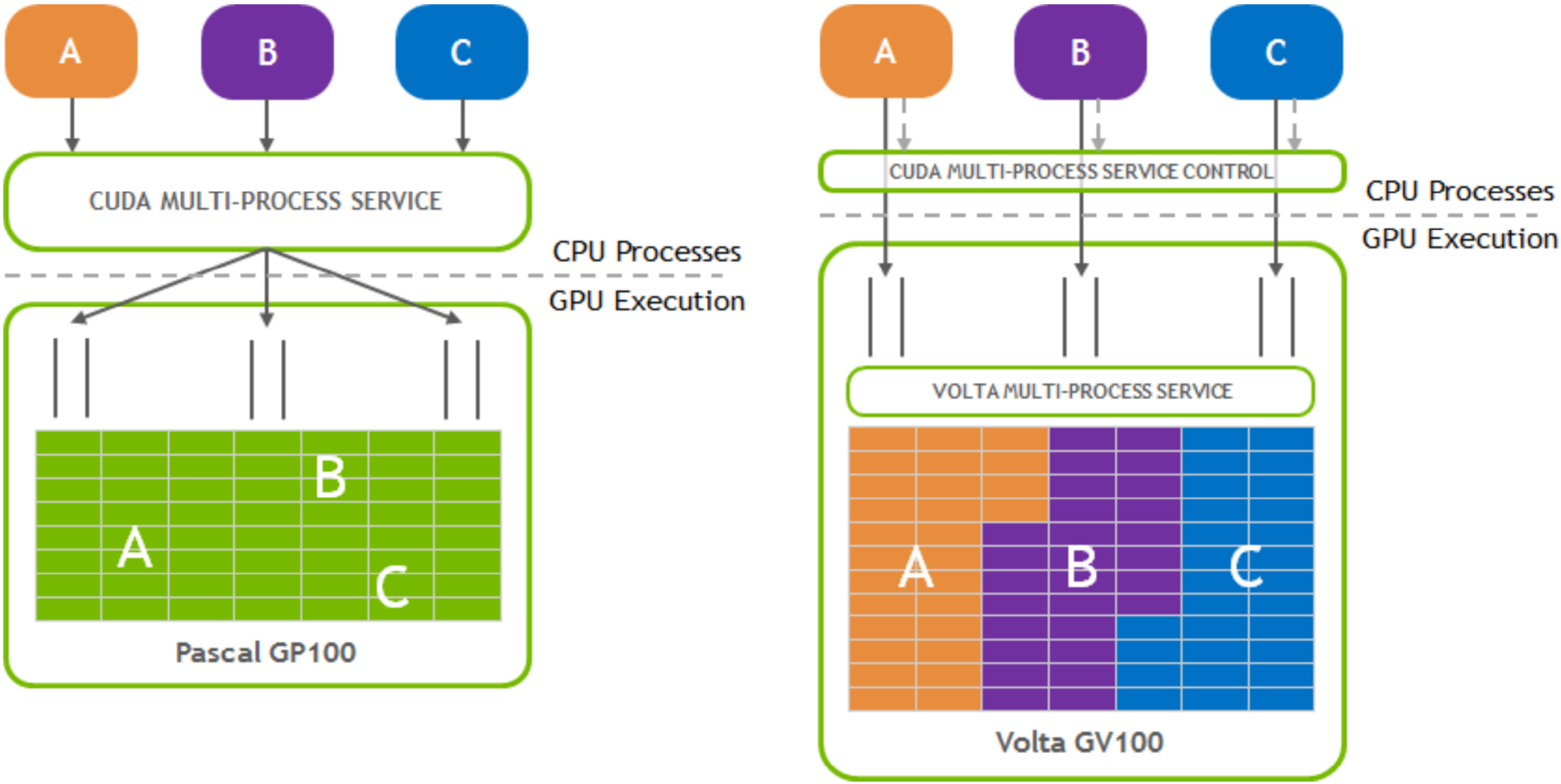
A new feature of the Volta GV100 architecture is the Volta Multi-Process Service (MPS) which allows for multiple applications to share the GPU, implemented with time slicing.

The MPS is for sharing the GPU amongst applications from a single user, not for multiple users.

The number of MPS clients increased from 16 on Pascal to 48 on Volta.

Volta provides very high throughput and low latency for deep learning inference when there is a batching system.

MPS on Pascal versus Volta



From the NVIDIA Tesla V100 GPU Architecture, NVIDIA 2017.

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

enumerating available devices

```
$ /tmp/count_devices
number of devices : 3
graphics card 0 :
  name : Tesla K20c
  number of multiprocessors : 13
graphics card 1 :
  name : GeForce GT 620
  number of multiprocessors : 2
graphics card 2 :
  name : Tesla K20c
  number of multiprocessors : 13
$
```

the program count_devices.cu

```
/* This program illustrates counting available devices,
 * compile it with nvcc and note the extension .cu
 * and for a more detailed version, see deviceQuery.cpp
 * of the GPU Computing SDK */

#include <stdio.h>

void printDeviceProp ( cudaDeviceProp p )
/*
 * prints some device properties */
{
   printf("  name : %s \n",p.name);
   printf("  number of multiprocessors : %d \n",
          p.multiProcessorCount);
}
```

the main program

```
int main ( int argc, char* argv[] )
{
   int deviceCount;
   cudaGetDeviceCount(&deviceCount);
   printf("number of devices : %d\n",deviceCount);

   for(int d = 0; d < deviceCount; d++)
   {
      cudaDeviceProp dev;
      cudaGetDeviceProperties(&dev,d);
      printf("graphics card %d :\n",d);
      printDeviceProp(dev);
   }

   return 0;
}
```

Concurrent Kernels and Multiple GPUs

1. Page Locked Host Memory
   - host memory that is page locked or pinned
   - executing a zero copy
2. Concurrent Kernels
   - streams and concurrency
   - squaring numbers with concurrent kernels
3. Using Multiple GPUs
   - counting the number of available devices
   - working with multiple GPUs

computing with multiple GPUs

Chapter 8 of the NVIDIA CUDA Best Practices Guide describes multi-GPU programming.

To work with p GPUs concurrently, the CPU can use

- p lightweight threads (Pthreads, OpenMP, etc); or
- p heavyweight threads (or processes) with MPI.

The command to select a GPU is `cudaSetDevice()`.

All inter-GPU communication happens through the host.

See the `simpleMultiGPU` of the GPU Computing SDK.