Advanced resources in CUDA

PRACE School

Barcelona Supercomputing Center. April, 16-20th, 2018

Manuel Ujaldón

Full Professor @ University of Malaga (Spain) Former CUDA Fellow @ NVIDIA Corporation (USA)







1. The read-only data cache (Kepler+) [4 slides]

- 2. Dynamic parallelism (Kepler+). [17]
 - 1. Data-dependent execution. [2]
 - 2. Recursive parallel algorithms. [4]
 - 3. Library calls from kernels. [3]
 - 4. Simplify the CPU/GPU division. [2]
- 3. Hyper-Q (Kepler+). [6]
- 4. GPU Boost (Kepler+). [6]
- 5. Unified memory. [19]
 - 1. Programming examples. [9]
 - 2. Summary. [1]
 - 3. Roadmap. [1]

6. Independent thread scheduling. [7]





1. The read-only data cache (Kepler+)





Differences in memory hierarchy: Fermi vs. Kepler



Kepler Memory Hierarchy





Motivation for using the new data cache

- Additional 48 Kbytes to expand L1 cache size.
- Highest miss bandwidth.

- Use the texture cache, but it is transparent to the programmer, and eliminates texture setup.
- Allows a global address to be fetched and cached, using a pipeline different from that of L1/shared.
- Flexible (does not require aligned accesses).
- Managed automatically by compiler.



How to use the new data cache

Annotate eligible kernel pointers with "const __restrict__"
 Compiler will automatically map loads to use read-only data cache path through texture memory.



A comparison with CUDA constant memory

To compare	Constant memory	Read-only data cache
Availability	From CUDA Compute Capability 1.0	From CCC 3.5 (available in CCC 1.0 using explicitly texture memory)
Size	64 Kbytes	48 Kbytes
Hardware implementation	A global memory partition (DRAM)	Texture cache expanding L1 (SRAM)
Access	Through an 8 Kbytes cache on each multiprocessor	Separated path in the graphics pipeline
Best feature	Very low latency	High data bandwidth
Worst feature	Lower bandwidth	Higher latency
Best scenario	Access with the same coefficient (do not involve threadIdx) to a small dataset	When the kernel is memory-bound even with a shared memory bandwidth already saturated



2. Dynamic parallelism (Kepler+)





What is dynamic parallelism?

The ability to launch new grids from the GPU:

Dynamically: Based on run-time data.

- Simultaneously: From multiple threads at once.
- Independently: Each thread can launch a different grid.





The way we did things in the pre-Kepler era: The GPU was a slave for the CPU

High data bandwidth for communications:

External: More than 10 GB/s (PCI-express 3).

O

Internal: More than 100 GB/s (GDDR5 video memory and 384 bits, which is like a six channel CPU architecture).





The way we do things in Kepler: GPUs launch their own kernels





Now programs run faster and are expressed in a more natural way.



Example 1: Dynamic work generation

Assign resources dynamically according to real-time demand, making easier the computation of irregular problems on GPU.

It broadens the application scope where it can be useful.





lower accuracy

Fine grid



Lower performance, higher accuracy Dynamic grid





Example 2: Deploying parallelism based on level of detail





Warnings when using dynamic parallelism

It is a much more powerful mechanism than it suggests from its simplicity in the code. However...

What we write within a CUDA kernel is replicated for **all** threads. Therefore, a kernel call will produce millions of launches if it is not used within an IF statement (which, for example, limits the launch to a single one from thread 0).

If a father block launches sons, can they use the shared memory of their father?

No. It would be easy to implement in hardware, but very complex for the programmer to guarantee the code correctness (avoid race conditions).





2. 1. Data-dependent execution





Data-dependent parallelism

The simplest possible parallel program:

- Loops are parallelizable.
- Workload is known at compile-time.

for (i=0; i<N; i++)
for (j=0; j<M; j++)
 convolution (i, j);</pre>



The simplest impossible parallel program:
 Workload is unknown at compile-time.
 The challenge is data partitioning.





Poor solution #1: Oversubscription. Poor solution #2: Serialization.



Now possible with dynamic parallelism: The two loops can be executed in parallel

The CUDA program for Kepler:

_global___ void convolution(int ElementsOnRow[])

for (j=0; j<ElementsOnRow[blockIdx]; j++) // Each block launches
 kernel <<< ... >>> (blockIdx, j) // ElementsOnRow[] kernels





2. 2. Recursive parallel algorithms





Recursive parallel algorithms prior to Kepler

Early CUDA programming model did not support recursion at all.

CUDA started to support recursive functions in version 3.1, but they can easily crash if the size of the arguments is large.

A user-defined stack in global memory can be employed instead, but at the cost of a significant performance penalty.

An efficient solution is possible using dynamic parallelism.



A simple example of parallel recursion: Quicksort

Typical divide-and-conquer algorithm hard to do on Fermi:

- Entire data-dependent execution.
- Recursively partition-and-sort data.



©.





CUDA code for quicksort

```
Low-performance version
                                                      High-performance version
global void qsort(int *data, int 1, int r)
                                              global void qsort(int *data, int 1, int r)
                                              Ł
  int pivot = data[0];
                                                int pivot = data[0];
  int *lptr = data+l, *rptr = data+r;
                                                int *lptr = data+l, *rptr = data+r;
 // Partition data around pivot value
                                                // Partition data around pivot value
 partition(data, l, r, lptr, rptr, pivot);
                                                partition(data, 1, r, lptr, rptr, pivot);
                                                // Use streams this time for the recursion
                                                cudaStream t s1, s2;
                                                cudaStreamCreateWithFlags(&s1, ...);
                                                cudaStreamCreateWithFlags(&s2, ...);
  // Launch next stage recursively
  int rx = rptr-data; lx = lptr-data;
                                                int rx = rptr-data; lx = lptr-data;
  if (1 < rx)
                                                if (1 < rx)
   qsort<<<...>>>(data,1,rx);
                                                  gsort<<<...,0,s1>>>(data,1,rx);
                                                if (r > lx)
  if (r > lx)
   gsort <<<...>>>(data,lx,r);
                                                  qsort << \ldots, 0, s_2 >>> (data, lx, r);
}
                                              }
```

left- and right-hand sorts are serialized

O

Use separate streams to achieve concurrency



Experimental results for Quicksort

- The lines of code were reduced in half.
- Performance was improved by 2x.





2. 3. Library calls from kernels





Programming model basics: CUDA run-time syntax & semantics





An example of simple library calls using cuBLAS (available from CUDA 5.0 on)





The father-child relationship in CUDA blocks





2. 4. Simplify the CPU/GPU division





A direct solver in matrix algebra: LU decomposition

Version for Fermi







Extended gains when our task involves thousands of LUs on different matrices

CPU-controlled work batching:

Serialize LU calls, or

©.

Face parallel P-threads limitations (10s).



Batching via dynamic parallelism:

Move top loops to GPU and launch 1000s of batches in parallel from GPU threads.





3. Hyper-Q (Kepler+)





Hyper-Q

In Fermi, several CPU processes can send thread blocks to the same GPU, but the concurrent execution of kernels was severely limited by hardware constraints.

- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
 - MPI processes, CPU threads (POSIX threads) or CUDA streams.
- This increments the % of temporal occupancy on the GPU.

FERMI 1 MPI Task at a Time



KEPLER 32 Simultaneous MPI Tasks







An example: 3 streams, each composed of 3 kernels

O

```
global kernel A(pars) {body} // Same for B...Z
                                                                 stream_1
   cudaStream t stream 1, stream 2, stream 3;
                                                                  kernel A
                                                                  kernel B
   cudaStreamCreatewithFlags(&stream 1, ...);
   cudaStreamCreatewithFlags(&stream_2, ...);
                                                                  kernel_C
   cudaStreamCreatewithFlags(&stream_3, ...);
                                                                 stream_2
  kernel A <<< dimgridA, dimblockA, 0, stream 1 >>> (pars);
stream
                                                                  kernel P
   kernel B <<< dimgridB, dimblockB, 0, stream 1 >>> (pars);
   kernel C <<< dimgridC, dimblockC, 0, stream 1 >>> (pars);
                                                                  kernel Q
                                                                  kernel R
N
   kernel P <<< dimgridP, dimblockP, 0, stream 2 >>> (pars);
stream
   kernel Q <<< dimgridQ, dimblockQ, 0, stream 2 >>> (pars);
                                                                 stream_3
   kernel R <<< dimgridR, dimblockR, 0, stream 2 >>> (pars);
                                                                  kernel X
m
   kernel X <<< dimgridX, dimblockX, 0, stream 3 >>> (pars);
stream
                                                                  kernel_Y
   kernel Y <<< dimgridY, dimblockY, 0, stream 3 >>> (pars);
                                                                  kernel Z
   kernel Z <<< dimgridZ, dimblockZ, 0, stream 3 >>> (pars);
```



Grid management unit: Fermi vs. Kepler

Fermi



©.

Kepler GK110





The relation between software and hardware queues







Manuel Ujaldon - Nvidia CUDA Fellow




4. GPU Boost (Kepler+)





What is the main contribution?

Allows to speed-up the GPU clock up to 17% if the power required by an application is low.

- The base clock will be restored if we exceed 235 W.
- We can set up a persistent mode which keep values permanently, or another one for a single run.





Every application has a different behaviour regarding power consumption

Here we see the average power (watts) on a Tesla K20X for a set of popular applications within the HPC field:





Those applications which are less power hungry can benefit from a higher clock rate

For the Tesla K40 case, 3 clocks are defined, 8.7% apart.



©.



GPU Boost compared to other approaches

It is better a stationary state for the frequency to avoid thermal stress and improve reliability.





Automatic clock switching



	Other vendors	Tesla K40
Default	Boost	Base
Preset options	Lock to base clock	3 levels: Base, Boost1 o Boost2
Boost interface	Control panel	Shell command: nv-smi
Target duration for boosts	Roughly 50% of run-time	100% of workload run time





GPU Boost - List of commands

Command	Effect	
nvidia-smi -q -d SUPPORTED_CLOCKS	View the clocks supported by our GPU	
nvidia-smi -ac <mem clock,<br="">Graphics clock></mem>	Set one of the supported clocks	
nvidia-smi -pm 1	Enables persistent mode: The clock settings are preserved after restarting the system or driver	
nvidia-smi -pm 0	Enables non-persistent mode: Clock settings revert to base clocks after restarting the system or driver	
nvidia-smi -q -d CLOCK	Query the clock in use	
nvidia-smi -rac	Reset clocks back to the base clock	
nvidia-smi -acp 0	Allow non-root users to change clock rates	



Example: Query the clock in use

onvidia-smi -q -d CLOCK -id=0000:86:00.0

_____NVSMI LOG______

Timestamp	:	Wed Jan 29 13:35:58 2014
Driver Version	:	319.37
Attached GPUs	:	5
GPU 0000:86:00.0		
Clocks		
Graphics	:	875 MHz
SM	:	875 MHz
Memory	:	3004 MHz
Applications Clocks		
Graphics	:	875 MHz
Memory	:	3004 MHz
Default Applications Clocks		
Graphics	:	745 MHz
Memory	:	3004 MHz
Max Clocks		
Graphics	:	875 MHz
SM	:	875 MHz
Memory	:	3004 MHz





5. Unified memory (Maxwell+)









In few years: All communications internal to the 3D chip





The new API: Accustom the programmer NOW to see the FUTURE memory





Performance sensitive to data proximity.





System requirements

	Required	Limitations		
CUDA version	Al least 6.0			
GPU	Kepler (GK10x+) or Maxwell (GM10x+)	Limited performance in CCC 3.0 and CCC 3.5		
Operating System	64 bits			
Windows	7 or 8	WDDM & TCC no XP/Vista		
Linux	Kernel 2.6.18+	All CUDA-supported distros, without ARM in earlier versions		
Linux on ARM	ARM64			
Mac OSX	Supported in CUDA 7			
	Manuel Uialdon - Nvidia CUDA Fellow			



Unified memory contributions

Simpler programming and memory model:

- Single pointer to data, accessible anywhere.
- Eliminate need for cudaMemcpy().
- Greatly simplifies code porting.
- Performance through data locality:
 - Migrate data to accessing processor.
 - Guarantee global coherency.
 - Still allows cudaMemcpyAsync() hand tuning.





CUDA memory types

Zero-Copy (pinned memory)		Unified Virtual Addressing	Unified Memory
CUDA call	cudaMallocHost(&A, 4);	cudaMalloc(&A, 4);	cudaMallocManaged(&A, 4);
Allocation fixed in	Main memory (DDR3)	Video memory (GDDR5)	Both
Local access for	CPU	Home GPU	CPU and home GPU
PCI-e access for	All GPUs	Other GPUs	Other GPUs
Other features	Avoid swapping to disk	No CPU access	On access CPU/GPU migration
Coherency	At all times	Between GPUs	Only at launch & sync.
Full support in	CUDA 2.2	CUDA 1.0	CUDA 6.0





Additions to the CUDA API

New call: cudaMallocManaged(pointer, size, flag)

- Drop-in replacement for cudaMalloc(pointer, size).
- The flag indicates who shares the pointer with the device:
 OcudaMemAttachHost: Only the CPU.
 - ©cudaMemAttachGlobal: Any other GPU too.
- All operations valid on device mem. are also ok on managed mem.
- New keyword: ___managed___
 - Global variable annotation combines with <u>device</u>.
 - Declares global-scope migratable device variable.
 - Symbol accessible from both GPU and CPU code.

New call: cudaStreamAttachMemAsync()

Manages concurrently in multi-threaded CPU applications.



Technical details

The maximum amount of unified memory that can be allocated is the **smallest** of the memories available on GPUs. Memory pages from unified allocations touched by CPU are required to **migrate back** to GPU before any kernel launch. The CPU cannot access any unified memory as long as GPU is executing, that is, a cudaDeviceSynchronize() call is required for the CPU to use unified memory. The GPU has exclusive access to unified memory when any kernel is executed on the GPU, and this holds even if the kernel does not touch the unified memory (see the first example in two slides).





5.1. Programming examples





Example 1: Access constraints (1)

```
device_____managed____int x, y = 2;
                                                // Unified memory
 _global___ void mykernel()
                                                 // GPU territory
  x = 10;
int main()
                                                 // CPU territory
 mykernel <<<1,1>>> ();
  y = 20; // ERROR: CPU access concurrent with GPU
  return 0;
```





Example 1: Access constraints (2)

```
device managed int x, y = 2;
                                               // Unified memory
 global void mykernel()
                                               // GPU territory
 x = 10;
                                               // CPU territory
int main()
 mykernel <<<1,1>>> ();
 cudaDeviceSynchronize();
                                                  // Problem fixed!
  // Now the GPU is idle, so access to "y" is OK
  y = 20;
  return 0;
```





Example 2: Increment a value "b" to all the N elements of an array "a"

CUDA code WITHOUT unified memory

CUDA code (from 6.0 on) WITH unified memory

<pre>global void incr (float *a, float b, int N) { int idx = blockIdx.x * blockDim.x + threadIdx.x; if (idx < N) a[idx] = a[idx] + b; } void main() {</pre>	<pre>global void incr (float *a, float b, int N) { int idx = blockIdx.x * blockDim.x + threadIdx.x; if (idx < N) a[idx] = a[idx] + b; } void main() {</pre>
unsigned int numBytes = N*sizeof(float);	
float* h_A = (float*) malloc(numBytes);	float* m_A; cudaMallocManaged(&m_A, numBytes);
<pre>float* d_A; cudaMalloc(&d_A, numBytes);</pre>	
<pre>cudaMemcpy(d_A,h_A,numBytes,cudaMemcpyHostToDevice);</pre>	
incr<< <n blocksize,blocksize="">>>(d_A,b,N);</n>	incr<< <n blocksize,blocksize="">>>(m_A,b,N);</n>
<pre>cudaMemcpy(h_A,d_A,numBytes,cudaMemcpyDeviceToHost);</pre>	cudaDeviceSynchronize();
cudaFree(d_A);	
free(h_A);	cudaFree(m_A);
}	}



Example 3: Sorting elements from a file. Now comparing to CPUs using C

CPU code in C	GPU code in CUDA (v. 6.0 on)	
<pre>void sortfile (FILE *fp, int N) { char *data; data = (char *) malloc(N);</pre>	<pre>void sortfile (FILE *fp, int N) { char *data; cudaMallocManaged(&data, N);</pre>	
<pre>fread(data, 1, N, fp);</pre>	<pre>fread(data, 1, N, fp);</pre>	
<pre>qsort(data, N, 1, compare);</pre>	<pre>qsort<<<>>>(data, N, 1, compare); cudaDeviceSynchronize();</pre>	
use_data(data);	use_data(data);	
<pre>free(data); }</pre>	<pre>cudaFree(data); }</pre>	

©.



Example 4: Cloning dynamic data structures WITHOUT unified memory

struct dataElem {
 int prop1;
 int prop2;
 char *text;

A "deep copy" is required:

We must copy the structure and everything that it points to. This is why C++ invented the copy constructor.

CPU and GPU cannot share a copy of the data (coherency). This prevents memcpy style comparisons, checksumming and other validations.







Cloning dynamic data structures WITHOUT unified memory (2)



```
void launch(dataElem *elem) {
   dataElem *g_elem;
   char *g text;
```

kernel<<< ... >>>(g elem);

```
int textlen = strlen(elem->text);
```

```
// Allocate storage for struct and text
cudaMalloc(&g_elem, sizeof(dataElem));
cudaMalloc(&g_text, textlen);
```



Cloning dynamic data structures WITH unified memory

CPU memory

Unified memory



GPU memory

void launch(dataElem *elem) { kernel<<< ... >>>(elem);

What remains the same:

- Data movement.
- GPU accesses a local copy of text.

• What has changed:

- Programmer sees a single pointer.
- CPU and GPU both reference the same object.
- There is coherence.
- To pass-by-reference vs. passby-value you need to use C++.





Example 5: Linked lists



- Almost impossible to manage in the original CUDA API.
- The best you can do is use pinned memory:
 - Pointers are global: Just as unified memory pointers.
 - Performance is low: GPU suffers from PCI-e bandwidth.
 - GPU latency is very high, which is critical for linked lists because of the intrinsic pointer chasing.



Linked lists with unified memory



Can pass list elements between CPU & GPU.

No need to move data back and forth between CPU and GPU.

Can insert and delete elements from CPU or GPU.

But program must still ensure no race conditions (data is coherent between CPU & GPU at kernel launch only).





Unified memory: Summary

Drop-in replacement for cudaMalloc() using cudaMallocManaged().

©cudaMemcpy() now optional.

Greatly simplifies code porting.

Less host-side memory management.

Enables shared data structures between CPU & GPU

Single pointer to data = no change to data structures.

Powerful for high-level languages like C++.



Unified memory evolution: Contributions on every abstraction level

Abstraction	Consolidated	Performed	Recent upgrades
level	in 2014	during 2015-16	(2017-2018)
High	Single pointer to data. No cudaMemcpy() is required	Prefetching mechanisms to anticipate data arrival in copies	System allocator unified
Medium	Coherence @ launch & synchronize	Migration hints	Stack memory unified
Low	Shared C/C++ data	Additional	Hardware-accelerated
	structures	OS support	coherence



6. Independent thread scheduling





A new model for synchronization and communication

In CUDA 9, we can define synchronization at 3 levels:

- Intra-warp: Flexible groups within a warp.
 - Lauch with a typical mykernel<<< , >>> or using cudaLaunchKernel();
- Inter-blocks: Multiple blocks within the grid.
 - Launch using cudaLaunchCooperativeKernel();
- Inter-GPUs: Multiple GPUs within the system.

 \bigcirc

Launch using cudaLaunchCooperativeKernelMultiDevice();





Intra-warp: Cooperative groups

Allows to define, synchronize and partition groups of cooperative threads within warps.

- Programs can be executed from Kepler on, but fast hardware infrastructure is included in Volta:
 - Program familiar algorithms and data structures in a natural way.
 Flexible thread grouping and synchronization
- Scalable execution (from a few threads to all running thrs.).

CUDA 9 provides a fully explicit synchronization model.

We must adapt legacy code for new execution model, removing implicit warp synchronous programming on all architectures. Example:
 CUDA 9 deprecates non-sync __shfl(), __ballot(), __any(), __all() as transition to new __shfl_sync(), __ballot_sync(), __any_sync(), ...











Example 1: Parallel sum reduction. Composable, robust and efficient





Example: Particle simulation

Without using cooperative groups:

// Threads update particles in parallel (pos, vel)
integrate<<<blocks, threads, 0, s>>>(particles);

// Note: implicit sync between kernel launches

// Build a regular grid spatial data structure to
// accelerate finding collisions between particles
collide<<<blocks, threads, 0, s>>>(particles);



Note that after integration and construction of the regular grid data structure, the ordering of particles in memory and mapping to threads changes, necessitating a synchronization between phases and multiple kernel launches. Using cooperative groups, all required synchronizations can be performed within a single kernel launch.





Whole-grid cooperation

Particle simulation update in a single kernel:

```
_global___ void particleSim(Particle *p, int N) {
```

```
grid_group g = this_grid();
```

```
for (i = g.thread_rank(); i < N; i += g.size())
integrate(p[i]);</pre>
```

```
g.sync(); // Sync whole grid!
```

```
for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);</pre>
```



Launch using cudaLaunchCooperativeKernel();

}



Multi-GPU cooperation

Large-scale multi-GPU simulation in a single kernel:

```
_global___ void particleSim(Particle *p, int N) {
```

```
multi_grid_group g = this_multi_grid();
```

```
for (i = g.thread_rank(); i < N; i += g.size())
integrate(p[i]);</pre>
```

```
g.sync(); // Sync all GPUs!
```

```
for (i = g.thread_rank(); i < N; i += g.size())
collide(p[i], p, N);</pre>
```

Launch using cudaLaunchCooperativeKernelMultiDevice();



}