



← Previous

(<https://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-cancer-detection/>)

Next →

(<https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/>)



Unified Memory in CUDA 6

Share:
(<https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/>)

Posted on November 18, 2013

(<https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/>) by

Mark Harris (<https://devblogs.nvidia.com/parallelforall/author/mharris/>)

77 Comments (https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/#disqus_thread)

Tagged C/C++ (<https://devblogs.nvidia.com/parallelforall/tag/cc/>), CUDA

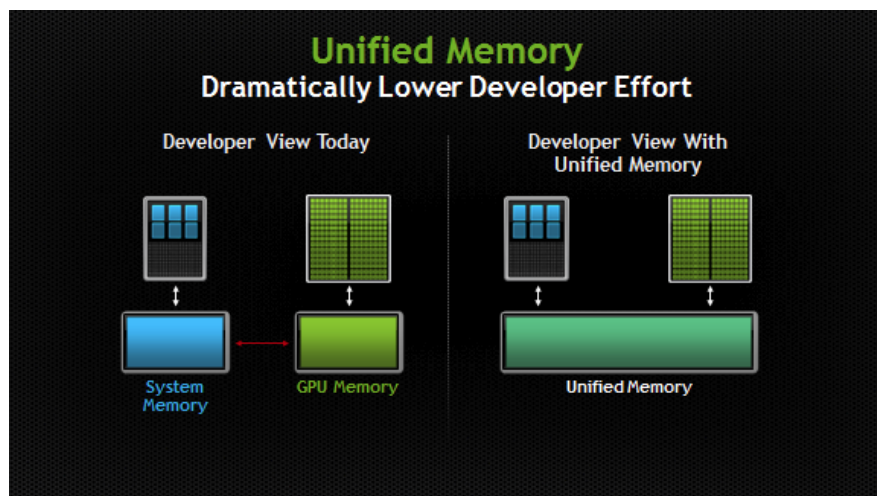
(<https://devblogs.nvidia.com/parallelforall/tag/cuda/>), CUDA 6

(<https://devblogs.nvidia.com/parallelforall/tag/cuda-6/>), Memory

(<https://devblogs.nvidia.com/parallelforall/tag/memory/>), Unified Memory

(<https://devblogs.nvidia.com/parallelforall/tag/unified-memory/>)


With CUDA 6, we're introducing one of the most dramatic programming model improvements in the history of the CUDA platform, Unified Memory. In a typical PC or cluster node today, the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus. Before CUDA 6, that is exactly how the programmer has to view things. Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the program. This adds a lot of complexity to CUDA programs.



Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically *migrates* data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.

In this post I'll show you how Unified Memory dramatically simplifies memory management in GPU-accelerated applications. The image below shows a really simple example. Both codes load a file from disk, sort the bytes in it, and then use the sorted data on the CPU, before freeing the memory. The code on the right runs on the GPU using CUDA and Unified Memory. The only differences are that the GPU version launches a kernel (and synchronizes after launching it), and allocates space for the loaded file in Unified Memory using the new API `cudaMallocManaged()`.

CPU Code	CUDA 6 Code with Unified Memory
<pre>void sortfile(FILE *fp, int N) { char *data; data = (char *)malloc(N); fread(data, 1, N, fp); qsort(data, N, 1, compare); use_data(data); free(data); }</pre>	<pre>void sortfile(FILE *fp, int N) { char *data; cudaMallocManaged(&data, N); fread(data, 1, N, fp); qsort<<<...>>>(data, N, 1, compare); cudaDeviceSynchronize(); use_data(data); cudaFree(data); }</pre>



If you have programmed CUDA C/C++ before, you will no doubt be struck by the simplicity of the code on the right. Notice that we allocate memory once, and we have a single pointer to the data that is accessible from both the host and the device. We can read directly into the allocation from a file, and then we can pass the pointer directly to a CUDA kernel that runs on the device. Then, after waiting for the kernel to finish, we can access the data again from the CPU. The CUDA runtime hides all the complexity, automatically migrating data to the place where it is accessed.

What Unified Memory Delivers

There are two main ways that programmers benefit from Unified Memory.

SIMPLER PROGRAMMING AND MEMORY MODEL

Unified Memory lowers the bar of entry to parallel programming on the CUDA platform, by making device memory management an optimization, rather than a requirement. With Unified Memory, now programmers can get straight to developing parallel CUDA kernels without getting bogged down in details of allocating and copying device memory. This will make both learning to program for the CUDA platform and porting existing code to the GPU simpler. But it's not just for beginners. My examples later in this post show how Unified Memory also makes complex data structures much easier to use with device code, and how powerful it is when combined with C++.

PERFORMANCE THROUGH DATA LOCALITY

By migrating data on demand between the CPU and GPU, Unified Memory can offer the performance of local data on the GPU, while providing the ease of use of globally shared data. The complexity of this functionality is kept under the covers of the CUDA driver and runtime, ensuring that application code is simpler to write. The point of migration is to achieve full bandwidth from each processor; the 250 GB/s of GDDR5 memory is vital to feeding the compute throughput of a Kepler GPU.

An important point is that a carefully tuned CUDA program that uses streams and `cudaMemcpyAsync` to efficiently overlap execution with data transfers may very well perform better than a CUDA program that only uses Unified Memory. Understandably so: the CUDA runtime never has as much information as the programmer

does about where data is needed and when! CUDA programmers still have access to explicit device memory allocation and asynchronous memory copies to optimize data management and CPU-GPU concurrency. Unified Memory is first and foremost a productivity feature that provides a smoother on-ramp to parallel computing, without taking away any of CUDA's features for power users.


Unified Memory or Unified Virtual Addressing?

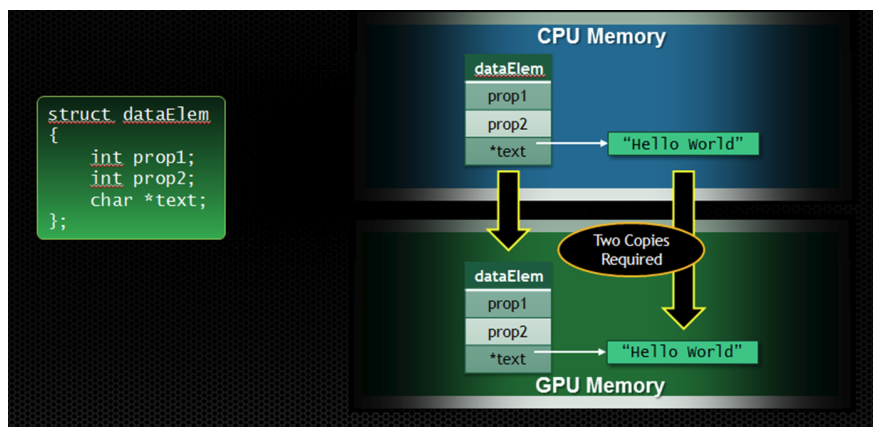
CUDA has supported Unified Virtual Addressing (UVA) (<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#unified-virtual-address-space>) since CUDA 4, and while Unified Memory depends on UVA, they are not the same thing. UVA provides a single virtual memory *address space* for all memory in the system, and enables pointers to be accessed from GPU code no matter where in the system they reside, whether its device memory (on the same or a different GPU), host memory, or on-chip shared memory. It also allows `cudaMemcpy` to be used without specifying where exactly the input and output parameters reside. UVA enables “Zero-Copy” memory, which is pinned host memory accessible by device code directly, over PCI-Express, without a `memcpy`. Zero-Copy provides some of the convenience of Unified Memory, but none of the performance, because it is always accessed with PCI-Express's low bandwidth and high latency.

UVA does not automatically migrate data from one physical location to another, like Unified Memory does. Because Unified Memory is able to automatically migrate data at the level of individual pages between host and device memory, it required significant engineering to build, since it requires new functionality in the CUDA runtime, the device driver, and even in the OS kernel. The following examples aim to give you a taste of what this enables.

Example: Eliminate Deep Copies

A key benefit of Unified Memory is simplifying the heterogeneous computing memory model by eliminating the need for deep copies when accessing structured data in GPU kernels. Passing data structures containing pointers from the CPU to the GPU requires doing a “deep copy”, as shown in the image below.





Take for example the following struct dataElem.

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

To use this structure on the device, we have to copy the struct itself with its data members, and then copy all data that the struct points to, and then update all the pointers in copy of the struct. This results in the following complex code, just to pass a data element to a kernel function.

```
void launch(dataElem *elem) {
    dataElem *d_elem;
    char *d_name;

    int namelen = strlen(elem->name) + 1;

    // Allocate storage for struct and name
    cudaMalloc(&d_elem, sizeof(dataElem));
    cudaMalloc(&d_name, namelen);

    // Copy up each piece separately, including new "name" pointer value
    cudaMemcpy(d_elem, elem, sizeof(dataElem), cudaMemcpyHostToDevice);
    cudaMemcpy(d_name, elem->name, namelen, cudaMemcpyHostToDevice);
    cudaMemcpy(&(d_elem->name), &d_name, sizeof(char*), cudaMemcpyHostToDevice);

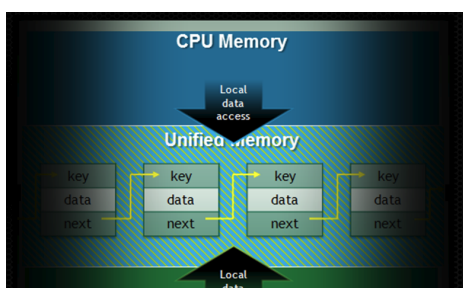
    // Finally we can launch our kernel, but CPU & GPU use different copies of "elem"
    Kernel<<< ... >>>(d_elem);
}
```

As you can imagine, the extra host-side code required to share complex data structures between CPU and GPU code has a significant impact on productivity. Allocating our dataElem structure in Unified Memory eliminates all the excess setup code, leaving us with just the kernel launch, which operates on the same pointer as the host code. That's a big improvement!

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

But this is not just a big improvement in the complexity of your code. Unified Memory also makes it possible to do things that were just unthinkable before. Let's look at another example.

Example: CPU/GPU Shared Linked Lists





Linked lists are a very common data structure, but because they are essentially nested data structures made up of pointers, passing them between memory spaces is very complex. Without Unified Memory, sharing a linked list between the CPU and the GPU is unmanageable. The only option is to allocate the list in Zero-Copy memory (pinned host memory), which means that GPU accesses are limited to PCI-express performance. By allocating linked list data in Unified Memory, device code can follow pointers normally on the GPU with the full performance of device memory. The program can maintain a single linked list, and list elements can be added and removed from either the host or the device.

Porting code with existing complex data structures to the GPU used to be a daunting exercise, but Unified Memory makes this so much easier. I expect Unified Memory to bring a huge productivity boost to CUDA programmers.

Unified Memory with C++

Unified memory really shines with C++ data structures. C++ simplifies the deep copy problem by using classes with copy constructors. A copy constructor is a function that knows how to create an object of a class, allocate space for its members, and copy their values from another object. C++ also allows the new and delete memory management operators to be overloaded. This means that we can create a base class, which we'll call Managed, which uses `cudaMallocManaged()` inside the overloaded new operator, as in the following code.

```
class Managed {
public:
    void *operator new(size_t len) {
        void *ptr;
        cudaMallocManaged(&ptr, len);
        cudaDeviceSynchronize();
        return ptr;
    }

    void operator delete(void *ptr) {
        cudaDeviceSynchronize();
        cudaFree(ptr);
    }
};
```

We can then have our String class inherit from the Managed class, and implement a copy constructor that allocates Unified Memory for a copied string.

```
// Deriving from "Managed" allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

public:
    // Unified memory copy constructor allows pass-by-value
    String (const String &s) {
        length = s.length;
        cudaMallocManaged(&data, length);
```

```

    cudaMemcpy(s.data, data, length, cudaMemcpyDeviceToHost);
    memcpy(data, s.data, length);
}

// ...
};

```

Likewise, we make our dataElem class inherit Managed.

```

// Note "managed" on this class, too.
// C++ now handles our deep copies
class dataElem : public Managed {
public:
    int prop1;
    int prop2;
    String name;
};

```

With these changes, the C++ classes allocate their storage in Unified Memory, and deep copies are handled automatically. We can allocate a dataElem in Unified Memory just like any C++ object.

```

dataElem *data = new dataElem;

```

Note that You need to make sure that every class in the tree inherits from Managed, otherwise you have a hole in your memory map. In effect, everything that you might need to share between the CPU and GPU should inherit Managed. You *could* overload new and delete globally if you prefer to simply use Unified Memory for everything, but this only makes sense if you have no CPU-only data because otherwise data will migrate unnecessarily.

Now we have a choice when we pass an object to a kernel function; as is normal in C++, we can pass by value or pass by reference, as shown in the following example code.

```

// Pass-by-reference version
__global__ void kernel_by_ref(dataElem &data) { ... }

// Pass-by-value version
__global__ void kernel_by_val(dataElem data) { ... }

int main(void) {
    dataElem *data = new dataElem;
    ...
    // pass data to kernel by reference
    kernel_by_ref<<<1,1>>>)(*data);

    // pass data to kernel by value -- this will create a copy
    kernel_by_val<<<1,1>>>(*data);
}

```

Thanks to Unified Memory, the deep copies, pass by value and pass by reference all just work. This provides tremendous value in running C++ code on the GPU.

The examples from this post are available on Github (<https://github.com/parallel-forall/code-samples/tree/master/posts/unified-memory>).

A Bright Future for Unified Memory

One of the most exciting things about Unified Memory in CUDA 6 is that it is just the beginning. We have a long roadmap of improvements and features planned around Unified Memory. Our first release is aimed at making CUDA programming easier, especially for beginners. Starting with CUDA 6, `cudaMemcpy()` is no longer a requirement. By using `cudaMallocManaged()`, you have a single pointer to data, and you can share complex C/C++ data structures between the CPU and GPU. This makes it much easier to write CUDA programs, because you can go straight to writing kernels, rather than writing a lot of data management code and maintaining duplicate host and device copies of all data. You are still free to use `cudaMemcpy()` (and particularly `cudaMemcpyAsync()`) for performance, but rather than a requirement, it is now an optimization.

Future releases of CUDA are likely to increase the performance of applications that use Unified Memory, by adding data prefetching and migration hints. We will also be adding support for more operating systems. Our next-generation GPU architecture will bring a number of hardware improvements to further increase performance and flexibility.

Find Out More

In CUDA 6, Unified Memory is supported starting with the Kepler GPU architecture (Compute Capability 3.0 or higher), on 64-bit Windows 7, 8, and Linux operating systems (Kernel 2.6.18+). To get early access to Unified Memory in CUDA 6, become a CUDA Registered Developer (<https://developer.nvidia.com/registered-developer-programs>) to receive notification when the CUDA 6 Toolkit (<https://developer.nvidia.com/cuda-toolkit>) Release Candidate is available. If you are attending Supercomputing 2013 in Denver this week, come to the NVIDIA Booth #613 (<http://www.nvidia.com/object/sc13.html>) and check out the GPU Technology Theatre to see one of my presentations about CUDA 6 and Unified Memory (Tuesday at 1:00 pm MTN, Wednesday at 11:30 am, or Thursday at 1:30 pm. Schedule here (<http://www.nvidia.com/object/sc13-technology-theater.html>)).

RELATED POSTS

The Saint on Porting C++ Classes to CUDA with Unified Memory
(<https://devblogs.nvidia.com/parallelforall/the-saint-porting-c-classes-cuda-unified-memory/>)

Unified Memory: Now for CUDA Fortran Programmers
(<https://devblogs.nvidia.com/parallelforall/unified-memory-cuda-fortran-programmers/>)

CUDA Pro Tip: Optimize for Pointer Aliasing (<https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimize-pointer-aliasing/>)

CUDACasts Episode 18: CUDA 6.0 Unified Memory
(<https://devblogs.nvidia.com/parallelforall/cudacasts-episode-18-cuda-6-0-unified-memory/>)

|| ∇

Share:

About Mark Harris



Mark is Chief Technologist for GPU Computing Software at NVIDIA. Mark has fifteen years of experience developing software for GPUs, ranging from graphics and games, to physically-based simulation, to parallel algorithms and high-performance computing. Mark has been using GPUs for general-purpose computing since before they even supported floating point arithmetic. While a Ph.D. student at UNC he recognized this nascent trend and coined a name for it: GPGPU (General-Purpose computing on Graphics Processing Units), and started GPGPU.org to provide a forum for those working in the field to share and discuss their work.

Follow @harrism on Twitter (https://twitter.com/intent/user?screen_name=harrism)
View all posts by Mark Harris →
(<https://devblogs.nvidia.com/parallelforall/author/mharris/>)

77 Comments Parallel Forall

 Login ▾

 Recommend 4  Share

Sort by Best ▾



Join the discussion...



Anton Murashov • 3 years ago

Hello. I just wonder is there any good use of UM to decrease latency of sending chunks of data to GPU for processing. Even those tens of microseconds needed to launch a kernel in some applications matter. CUDA kernels can be pre-launched and spin on host pinned memory (waiting for data to process), but that means each loop condition check is PCIe 'read' transaction. In contrast UM pages 'touched' by CPU willing to give some data for GPU to process could theoretically be invalidated/updated by PCIe 'posted write', which is more efficient than enormous number of 'reads' over PCIe from device to host memory. You say that UM is synchronized on kernel launch if CPU has touched it - so I assume the answer to my question is 'no', but I just want to check if my understanding is correct. Thank you!

7 ^ | v • Reply • Share ›



Peter V./Vienna/Austria/Europe • 2 years ago

Hi,

according to this document

<http://on-demand.gputechcon...>

page 37 "CPU cannot access memory while GPU is accessing it":

I have a heavy multi threaded program which crashed all the time. Then I found the reason: It crashed because I accessed the Unified pointer in one thread via CPU, while it was accessed in other thread via GPU.

My question is this:

Does it mean that any Unified pointer must have an exclusive access either by the CPU or by the GPU? I mean this: I allocate 50 Unified pointers which different address and each pointer is accessed only in one thread.

But apparently all pointers are blocked.

Can you please confirm me this behaving?

Regards, Peter

Attached the picture with 16 Unified addresses - one address per thread.

[see more](#)

1 ^ | v • Reply • Share ›



Mark Harris Mod ➔ Peter V./Vienna/Austria/Europe • 2 years ago

By default it is assumed that any managed allocation could be accessed from any active kernel and therefore access from the CPU is disallowed while kernels are running on the GPU. But you can achieve finer grain control by using CUDA streams (other than the default/NULL stream) and `cudaStreamAttachMemAsync()`. See Appendix J of the CUDA C Programming Guide: <http://docs.nvidia.com/cuda...>

^ | v • Reply • Share ›



Peter V./Vienna/Austria/Europe ➔ Mark Harris • 2 years ago

I changed my code according to the link you provided me. I have now this issue: When running under heavy load [~150 threads which really run parallel] then there is a very rare and not reproduce able situation that some data from the ~2 MB big buffer is not copied correctly from the allocated Managed Memory to CPU memory [I do it for performance reasons].

What I can say is it has something to do with the high amount of concurrent running threads but that far I don't see any mistake in the code and especially the wrong data is just a buffer with a constant value which never changes.

Do you have any idea/hint what it can be or are 150 concurrent running threads on a QUADRO K5000 too much? Each thread uses ~25k bytes of shared memory but when I understand it correctly the card has only 64 KB of shared memory. Is there a "swapping" and when yes, where?

^ | v • Reply • Share ›



Mark Harris Mod ➔ Peter V./Vienna/Austria/Europe • 2 years ago

First, 150 threads running on a GPU is a very small number of threads -- you need thousands of threads active to keep a GPU like a K5000 busy.

Second, without seeing any code, I can't surmise what could be the cause of the failed data update in your app based on the description. My

cause of the failed data update in your app based on the description. My suggestion would be for you to simplify it to the simplest possible example that demonstrates the problem (reliably), and then post a question on devtalk.nvidia.com (or StackOverflow under the cuda tag).

Third: shared memory is not virtualized. If you use 25KB per thread, and launch thread blocks with more than 1 thread, then your launch should fail due to insufficient resources (each thread block can allocate at most 48KB of shared memory on a K5000, IIRC). Are you checking errors properly to make sure your kernel is actually running and not failing to launch and reporting a CUDA error? If you are only running one thread per thread block, then you will be VASTLY underutilizing the computational resources on your GPU. Shared memory isn't really intended to be used for per-thread data: it's called "shared" for a reason.

[see more](#)

^ | v • Reply • Share ›



Peter V./Vienna/Austria/Europe → Mark Harris • 2 years ago

I was unfortunately inexact in my previous question:
I have 150 CPU threads and each of them starts a stream with 150 blocks of 149 threads and each block uses a static array in shared memory of 1710 bytes.
This means that in total we have ~22k threads split into streams/blocks running concurrently. I use Unified Memory for allocation and assign this allocated memory to the stream. That far it works --> no segmentation fault as I had it before without assigning it to a stream.

Can this be now a bit too heavy for the Quadro K5000?
The problem with the wrong copied data ["readonly" which is not changed via GPU; it is just read and then written back as a reference] starts to occur with threads > 100 CPU threads * 150 block * 149 threads...

To extract the code to the simplest possible example is very hard therefore I want to clarify if I am not overloading the GPU. Should I restrict the concurrently running CPU threads to 50?

^ | v • Reply • Share ›



Mark Harris Mod → Peter V./Vienna/Austria/Europe • 2 years ago

I think this is too complex to debug in a comment stream. I suggest you start a thread on devtalk.nvidia.com, or contact me directly.

Couple of comments. I assume you mean launching 1 block from each of 150 CPU threads. Only 32 separate streams can run kernels concurrently on Kepler GPUs, so having more than 32 CPU threads launching kernels is not really efficient. Also, thread blocks should be a multiple of 32 threads in size for best efficiency (the warp size).

You might want to read the CUDA best practices guide and Kepler Tuning Guide.

In any case, it sounds like a bug (either in your program or in the CUDA runtime), so it would be good to learn more on a higher-efficiency forum.

^ | v • Reply • Share ›



Peter V./Vienna/Austria/Europe → Mark Harris • 2 years ago

Hi Mark,

solved! I was never doubting CUDA runtime has a bug. It is quite embarrassing for me:

I

used in a heavy multi threaded C++ program the strtok function instead of strtok_r. Due this stupid mistake in very rare situations the buffer was not filled correctly!

How ever, because for my algorithm it is not really needed to use the Managed Memory feature I switched to the cudaMalloc() function. I limit the amount of maximal GPU threads to the warp

size of the GPU [32].

Now I can say after extensively tests it runs perfectly.

Mark, I admire your support here and that you find time to answer all these questions!

All the best for you and your family 2015!

2 ^ | v • Reply • Share ›



Anderson • 3 years ago

Good night,

I wonder if there are any scheduled for launch cuda 6 for registered users to date. I've seen that my vga is compatible (gt 640m).

1 ^ | v • Reply • Share ›



Mark Harris Mod → Anderson • 3 years ago

The CUDA 6 Release Candidate is available now! <https://developer.nvidia.co...>

1 ^ | v • Reply • Share ›



Adam MacDonald • 3 years ago

I've written a system for abstracting memory copies into my API, so the user can just use his data on the CPU and GPU seamlessly, using a checksum internally to determine if anything has changed and only transferring as late as necessary. Every part of the API is made more complex because of this. I'm really looking forward to just deleting all of that logic.

1 ^ | v • Reply • Share ›



Det → Adam MacDonald • 3 years ago

Yeah, but it's still only supported from Kepler onwards.

^ | v • Reply • Share ›



mpeniak → Det • 3 years ago

which is great, I wouldn't want to use old cards seeing what the new post-fermi hw can do

1 ^ | v • Reply • Share ›



terry spitz • 3 years ago

great functionality, definitely a move in the right direction for allowing porting existing code rather than rewriting. can we expect virtual function table rewiring for true C++ object copying to device? any support for STL on device (start with vector, shared_ptr) - even just read-only?

1 ^ | v • Reply • Share ›



Mark Harris Mod → terry spitz • 3 years ago

The problem with virtual function tables is that AFAIK the C++ standard does not specify the format/layout/implementation of vtables. This makes it nearly impossible to support calling virtual functions on shared objects across all C++ host compilers supported by CUDA / nvcc. As for STL, that is something that we intend to look at, but nothing I can share here yet.

1 ^ | v • Reply • Share ›



Mark Harris Mod • 3 years ago

We will be rolling out Unified Memory for additional languages and platforms in future releases of CUDA (and CUDA Fortran).

1 ^ | v • Reply • Share ›



Fortran • 3 years ago

As someone who works with CUDA Fortran, I am hoping the day comes soon when NVIDIA/PGI Fortran also includes a similar functionality. I'd really like to get rid of all those freaking cudaMemcpy's in my code!

1 ^ | v • Reply • Share ›



Mark Harris Mod → Fortran • 3 years ago

As of PGI 14.7, Unified Memory is available for CUDA Fortran programmers too!

<http://devblogs.nvidia.com/...>

^ | v • Reply • Share ›



Zach • 14 days ago

Very helpful post—thank you. One question: In the linked list section, you said “The program can maintain a single linked list, and list elements can be added and removed from either the host or the device.” I am writing a program to do precisely this: initialize a linked list on the host and manipulate it on the device (by allocating/rearranging elements on the device). I am having my data elements inherit from the “Managed” class as you have done. This approach works fine on the host; however, when trying to allocate elements from the device, I get the error that calling a host function (Managed::operator new) from a global function is not allowed.

How can I allocate elements from the device and add to the list? I thought about making a device version of the Managed constructor, but I don't know how/if this is even possible.

^ | v • Reply • Share ›



駿延何 • a year ago

Hi,
I wonder whether multi-level pointer is supported on unified memory?

^ | v • Reply • Share ›



Mark Harris Mod ➔ 駿延何 • a year ago

Do you mean pointers to pointers? In CUDA, pointers are just pointers, so yes, this works.

^ | v • Reply • Share ›



Mehmed • 2 years ago

What about the unified memory first Maxwell had in the roadmap and then Pascal got? Is it something else? Hardware based unified memory? Will Pascal get it or has it been delayed again and now Volta gets it?

^ | v • Reply • Share ›



scurvyydog • 2 years ago

Can't wait to see this implemented in blenders cycles rendering.

^ | v • Reply • Share ›



Sergi • 2 years ago

Hello Mark,

in `cuda-7.0/samples/0_Simple/UnifiedMemoryStreams/` the example uses `cudaDeviceSynchronize()` in the construction and destruction. The documentation mentions also when it is necessary and your full github in <https://github.com/parallel...> follows it and `cudaDeviceSynchronize()` is used exclusively right after kernel invocation.

My question is whether there would be any good reason to include `cudaDeviceSynchronize()` in your Managed base class right after `cudaMallocManaged` and before `cudaFree`. `cudaDeviceSynchronize` call is much shorter (<1%) than `cudaMallocManaged`, so there is no real penalty in performance. But, probably, that's not the main argument to do so or not.

^ | v • Reply • Share ›



Mark Harris Mod ➔ Sergi • 2 years ago

Basically, the programming model assumes that any kernel launched can be accessing any managed memory attached to the “global” stream, even if that memory was allocated `_after_` the kernel was launched. This means that if you want to allocate managed memory and access on the CPU right away, you have to either make sure that all kernels have been synchronized OR you have to attach to the “host” stream when you allocate (i.e. do `cudaMallocManaged(&ptr, size, cudaMemAttachHost)`). The latter choice then requires that the data be attached to “global” or a specific stream if it needs to be accessed from the GPU.

^ | v • Reply • Share ›



Nitesh Upadhyay • 2 years ago

Hello Mark

This shared memory is provided by the hardware (Keplar) or only software abstraction? Would this not work on Tesla or Fermi?

^ | v • Reply • Share ›



Mark Harris Mod → Nitesh Upadhyay • 2 years ago

Shared memory is something specific and different in CUDA (it is a small on-chip memory in the GPU SM that threads can share data in). On Kepler, Unified Memory is a software abstraction with hardware support. It is not supported on earlier architectures.

^ | v • Reply • Share ›



Nitesh Upadhyay → Mark Harris • 2 years ago

By shared I meant shared between GPU and CPU. I'm aware about normal shared memory in gpu which is shared in block.

In this paper <http://dedis.cs.yale.edu/20...> they talk about cpu-gpu shared memory in cuda -5.0.

which I didn't find any where in documentation of cuda.

Is this unified memory is accessible from both cpu and gpu directly?

^ | v • Reply • Share ›



Sergi • 2 years ago

Hi Mike,

what do you think about: <https://www.bu.edu/caadlab/...> ?

I have a project where I'd like to use UMA if it really pays off. From that analysis, I see the best improvements, when happen, are modest versus non-UMA. In my personal case, I have a large set of data, but each thread only needs to access a bit of it and solve small (<20x20) linear problems, but many of them > 1e9 total. I had thought of using dynamic parallelism for some of those where the condition number is better to go ahead with the next bunch. So, dynamic parallelism and UMA seemed to me the way. But ... I am a bit confused now.

Thanks,

Sergi

^ | v • Reply • Share ›



Mark Harris Mod → Sergi • 2 years ago

Not sure who Mike is, and it's Unified Memory, not UMA (UMA is something specific -- related, but different). Anyway, it does sound like your problem may be amenable to both Unified Memory and Dynamic Parallelism.

^ | v • Reply • Share ›



Sergi → Mark Harris • 2 years ago

Hi _Mark_,

sorry about the name. I have been trying to find the answer by myself reading quite a few documents and posts last days and for some reason I failed in my first post here.

By UMA, I am sure we both understand Unified Memory Access. I had understood you need UMA to make use of Unified Memory. The abstract in that paper seems quite clear in the relationship with Unified Memory in this post and `cuda>6`. The rest of the paper seems quite respectable too. What I would like to know is whether you agree with them and the apparent slow down effects or it can be mitigated. (NB: I have gone ahead with Unified Memory in my project with a K20m).

Thanks

^ | v • Reply • Share ›



Mark Harris Mod → Sergi • 2 years ago

The problem is that UMA is already an existing acronym, which a different meaning from the way the authors of that paper use it. UMA stands for *UNIFORM* Memory Access, not Unified Memory Access. The Uniform refers to the performance of memory accesses -- it means that access time is independent of which processor makes the access. See <http://en.wikipedia.org/wik...>

This is definitely not the same as Unified Memory.

The paper is otherwise reasonable. As we have said in the past, the initial versions of Unified Memory are aimed at simplifying heterogeneous programming, making it easier to port applications especially with complex data structures. Future releases (and hardware) will continue to improve this ease of use and improve performance.

^ | v • Reply • Share ›



Dinesh • 2 years ago

Is data automatically moved to shared memory when such optimization is possible?

^ | v • Reply • Share ›



Mark Harris Mod → Dinesh • 2 years ago

No, not currently.

^ | v • Reply • Share ›



Dinesh → Mark Harris • 2 years ago

Thanks

^ | v • Reply • Share ›



Usman Shahid • 3 years ago

Does GTX 750ti support UVM? If not... would it later?

^ | v • Reply • Share ›



Mark Harris Mod → Usman Shahid • 3 years ago

Yes, Unified Memory is supported on Kepler GK110 and later GPUs. I believe GeForce GTX 750ti is based on a Maxwell GM107 GPU.

^ | v • Reply • Share ›



Usman Shahid → Mark Harris • 3 years ago

May be my fault, but after hard struggle my 750ti is unable to execute UVM code. I did some search on internet and found on some sites that 750ti doesn't have native support to UVM. Is it true? Kindly guide me where am I wrong. I am using VS 2013 & CUDA 6.5 with latest drivers

^ | v • Reply • Share ›



Mark Harris Mod → Usman Shahid • 3 years ago

Is your application 64-bit? If you compile a 32-bit application, it will not have access to Unified Memory. Per the CUDA Programming guide, Unified Memory requires a 64-bit host application and operating system. (Linux or Windows)

^ | v • Reply • Share ›



Yu-Chen Wu • 3 years ago

Hello, I want to know if the GPUs which support UVA will keep cache coherence between CPU and GPU at running time(when the kernel function is running). Thank you.

^ | v • Reply • Share ›



Mark Harris Mod → Yu-Chen Wu • 2 years ago

No, current GPUs cannot do this.

^ | v • Reply • Share ›



Taegeun Kang • 3 years ago

GTX 660 does support unified memory?

^ | v • Reply • Share ›



James Dang • 3 years ago

For the C++ code, you should note that you have to implement operator new[] and operator delete[] as well if you want arrays of a non-primitive type. Took a few hours struggling with segfaults to figure that out.

^ | v • Reply • Share ›



RD • 3 years ago

Thanks for this great function.

Is there any complete example code available? (especially one that uses the recent class

is there any complete example code available ? (especially one that uses the parent class Managed)

^ | v • Reply • Share ›



Coiby • 3 years ago

Is "elem->name" a typo?

dataElem is defined as:

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

You should use elem->text instead.

^ | v • Reply • Share ›



Mark Ebersole → Coiby • 3 years ago

Good catch Coiby! I'll change the "char *text" in dataElem to "char *name".

^ | v • Reply • Share ›



Frank Winter • 3 years ago

Our application framework allows the user to access a given data portion from both the CPU and the GPU. In order to provide a high performance in either case we employ different data layouts depending on the processor type that makes the access, e.g. AoS vs. SoA. Thus, we change the data layout on the fly when migrating data between GPU and CPU memory domains.

Now, since unified memory does the data migration for you I guess it's job is done by just copying the data. Thus I assume it's not possible to manage-allocate a chunk of memory and pass a user-defined data layout transformation function to the malloc call. I am talking about a optional software hook that would get called when data was migrated by the driver/manager into a 'staging area' and from there it could be processed and stored into it's final destination by a user-defined function.

Such a feature would be nice to have.

^ | v • Reply • Share ›



Mark Harris Mod → Frank Winter • 3 years ago

Unified Memory migration is at the page level. It would be very difficult to generically handle user-defined memory transformations like you describe at that level. I don't know of any CPU allocators that apply transformations, for example. If it requires explicit memcopies anyway, then Unified Memory doesn't gain you much.

As I pointed out in the article, it's going to be difficult for an automatic system like this to outperform custom-tuned memory management frameworks like you describe -- the programmer usually has more information than the runtime or compiler. Since you already have a framework, there is no reason you can't keep using it.

^ | v • Reply • Share ›



Frank Winter • 3 years ago

You say: 'Don't think about unified memory as a page cache on the device.'

I think I have to disagree here. This is exactly what you should think if you're referring to how things work under the hood. Otherwise I would be very surprised. Let me make the point. The virtual address space available to the CPU is much bigger than the physical memory on the GPU. (Let's forget for a moment that all address spaces are mapped into one UVA space.) Let's make an example: Suppose we are talking about a K20 with 6 GB and a total of 48 GB CPU memory. Let's further assume one manage-allocates 10 chunks of 1 GB each. First question here: You say the default memory location is the GPU. What happens when allocating the 7th chunk? Does the first chunk get copied to CPU memory? Do we have an 'out of memory' error? Or is really a 'first-touch allocation' mechanism at work?

Okay, let's say the allocation of 10 chunks was successful. Now, suppose the user launches 10 kernels sequentially each using a different memory chunk: first kernel uses chunk 1, second kernel uses chunk 2, etc. I understand that unified memory leaves the memory on the device after a kernel launch. Thus, before launching the 7th kernel we have 6 chunks used by the previous

[see more](#)

^ | v • Reply • Share ›



Mark Harris Mod → Frank Winter • 3 years ago

On current hardware, Unified Memory does not allow oversubscribing GPU memory. You are limited on managed allocations to the amount of memory available on the GPU (smallest memory if there are multiple GPUs). Future hardware may support GPU page faulting which will enable us to oversubscribe GPU memory, and to do what you describe. In your example, I believe the allocations should start failing after you exceed available GPU memory. Today, pages are not migrated out of GPU memory unless they are touched by the CPU.

1 ^ | v • Reply • Share ›

[Load more comments](#)

ALSO ON PARALLEL FORALL

New Compiler Features in CUDA 8

1 comment • 5 months ago•



Pioneer American — I'd love a nice work-stealing task library like Intel's Tthreading Building Blocks on ...

Personalized Aesthetics: Recording the Visual Mind ...

1 comment • 12 days ago•



teddyknox — I'm having fun wading through the pedantic pseudo-analysis here. The topic is similar ...

Exploring the SpaceNet Dataset Using DIGITS

11 comments • 7 months ago•



Jon Barker — Here's an example of how to apply a Euclidean distance transform to a bitmap image: ...

DetectNet: Deep Neural Network for Object Detection in DIGITS

17 comments • 8 months ago•



Alper — I wrote a small macro tool for faster labeling of custom detectnet datasets. It allows you ...

[Subscribe](#) [Add Disqus to your site](#) [Add Disqus](#) [Add](#) [Privacy](#)