Managing Accelerated Application Memory with CUDA C/C++ Unified Memory and nvprof

The *CUDA Best Practices Guide* (http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations), a highly recommended followup to this and other CUDA fundamentals labs, recommends a design cycle called **APOD**: **A**ssess, **P**arallelize, **O**ptimize, **D**eploy. In short, APOD prescribes an iterative design process, where developers can apply incremental improvements to their accelerated application's performance, and ship their code. As developers become more competent CUDA programmers, more advanced optimization techniques can be applied to their accelerated codebases.

This lab will support such a style of iterative development. You will be using the **NVIDIA Command Line Profiler** to qualitatively measure your application's performance, and to identify opportunities for optimization, after which you will apply incremental improvements before learning new techniques and repeating the cycle. As a point of focus, many of the techniques you will be learning and applying in this lab will deal with the specifics of how CUDA's **Unified Memory** works. Understanding Unified Memory behavior is a fundamental skill for CUDA developers, and serves as a prerequisite to many more advanced memory management techniques.

Prerequisites

To get the most out of this lab you should already be able to:

- Write, compile, and run C/C++ programs that both call CPU functions and launch GPU kernels.
- Control parallel thread hierarchy using execution configuration.
- Refactor serial loops to execute their iterations in parallel on a GPU.
- Allocate and free Unified Memory.

Objectives

By the time you complete this lab, you will be able to:

- Use the **NVIDIA Command Line Profiler** (**nprof**) to profile accelerated application performance.
- Leverage an understanding of **Streaming Multiprocessors** to optimize execution configurations.
- Understand the behavior of **Unified Memory** with regard to page faulting and data migrations.
- Use **asynchronous memory prefetching** to reduce page faults and data migrations for increased performance.
- Employ an iterative development cycle to rapidly accelerate and deploy applications.

Iterative Optimizations with the NVIDIA Command Line Profiler

The only way to be assured that attempts at optimizing accelerated code bases are actually successful is to profile the application for quantitative information about the application's performance. `nvprof` is the NVIDIA command line profiler. It ships with the CUDA toolkit, and is a powerful tool for profiling accelerated applications.

`nvprof` is easy to use. Its most basic usage is to simply pass it the path to an executable compiled with `nvcc`. `nvprof` will proceed to execute the application, after which it will print a summary output of the application's GPU activities, CUDA API calls, as well as information about **Unified Memory** activity, a topic which will be covered extensively later in this lab.

When accelerating applications, or optimizing already-accelerated applications, take a scientific and iterative approach. Profile your application after making changes, take note, and record the implications of any refactoring on performance. Make these observations early and often: frequently, enough performance boost can be gained with little effort such that you can ship your accelerated application. Additionally, frequent profiling will teach you how specific changes to your CUDA codebases impact its actual performance: knowledge that is hard to acquire when only profiling after many kinds of changes in your codebase.

Exercise: Profile an Application with nvprof

01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) (<------ you can click on this and any of the source file links in this lab to open them for editing) is a naively accelerated vector addition program. Use the two code execution cells below (by `CTRL` + clicking them). The first code execution cell will compile (and run) the vector addition program. The second code execution cell will profile the executable that was just compiled using `nvprof`.

After profiling the application, answer the following questions using information displayed in the profiling output:

- What was the name of the only CUDA kernel called in this application?
- How many times did this kernel run?
- How long did it take this kernel to run? Record this time somewhere: you will be optimizing this application and will want to know how much faster you can make it.

In [ ]:
```
!nvcc -arch=sm_70 -o single-thread-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nvprof ./single-thread-vector-add
```

Exercise: Optimize and Profile

Take a minute or two to make a simple optimization to 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) by updating its execution configuration so that it runs on many threads in a single thread block. Recompile and then profile with `nvprof` using the code execution cells below. Use the profiling output to check the runtime of the kernel. What was the speed up from this optimization? Be sure to record your results somewhere.

In [ ]:
```
!nvcc -arch=sm_70 -o multi-thread-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nvprof ./multi-thread-vector-add
```

Exercise: Optimize Iteratively

In this exercise you will go through several cycles of editing the execution configuration of 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu), profiling it, and recording the results to see the impact. Use the following guidelines while working:

- Start by listing 3 to 5 different ways you will update the execution configuration, being sure to cover a range of different grid and block size combinations.
- Edit the 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) program in one of the ways you listed.
- Compile and profile your updated code with the two code execution cells below.
- Record the runtime of the kernel execution, as given in the profiling output.
- Repeat the edit/profile/record cycle for each possible optimzation you listed above

Which of the execution configurations you attempted proved to be the fastest?

```
In [ ]: !nvcc -arch=sm_70 -o iteratively-optimized-vector-add 01-vector-add/01-vector-add.cu -run
```

```
In [ ]: !nvprof ./iteratively-optimized-vector-add
```

Streaming Multiprocessors and Querying the Device

This section explores how understanding a specific feature of the GPU hardware can promote optimization. After introducing **Streaming Multiprocessors**, you will attempt to further optimize the accelerated vector addition program you have been working on.

The following slides present upcoming material visually, at a high level. Click through the slides before moving on to more detailed coverage of their topics in following sections.

In [1]:
```
%%HTML

<div align="center"><iframe src="https://docs.google.com/presentation/d/e/2PACX-1vQTzaK1iaFkxgYxaxR5QgHCVx1Z
```

Streaming Multiprocessors and Warps

The GPUs that CUDA applications run on have processing units called **streaming multiprocessors**, or **SMs**. During kernel execution, blocks of threads are given to SMs to execute. In order to support the GPU's ability to perform as many parallel operations as possible, performance gains can often be had by *choosing a grid size that has a number of blocks that is a multiple of the number of SMs on a given GPU.*

Additionally, SMs create, manage, schedule, and execute groupings of 32 threads from within a block called **warps**. A more in depth coverage of SMs and warps (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation) is beyond the scope of this course, however, it is important to know that performance gains can also be had by *choosing a block size that has a number of threads that is a multiple of 32.*

Programmatically Querying GPU Device Properties

In order to support portability, since the number of SMs on a GPU can differ depending on the specific GPU being used, the number of SMs should not be hard-coded into a codebase. Rather, this information should be acquired programatically.

The following shows how, in CUDA C/C++, to obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

```
int deviceId;
cudaGetDevice(&deviceId);                  // `deviceId` now points to the id of the currently activ
e GPU.

cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId); // `props` now has many useful properties about
                                           // the active GPU device.
```

Exercise: Query the Device

Currently, `01-get-device-properties.cu` (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/04-device-properties/01-get-device-properties.cu) contains many unassigned variables, and will print gibberish information intended to describe details about the currently active GPU.

Build out `01-get-device-properties.cu` (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/04-device-properties/01-get-device-properties.cu) to print the actual values for the desired device properties indicated in the source code. In order to support your work, and as an introduction to them, use the CUDA Runtime Docs (http://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html) to help identify the relevant properties in the device props struct. Refer to the solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/04-device-properties/solutions/01-get-device-properties-solution.cu) if you get stuck.

In [ ]:
```
!nvcc -arch=sm_70 -o get-device-properties 04-device-properties/01-get-device-properties.cu -run
```

Exercise: Optimize Vector Add with Grids Sized to Number of SMs

Utilize your ability to query the device for its number of SMs to refactor the `addVectorsInto` kernel you have been working on inside 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) so that it launches with a grid containing a number of blocks that is a multiple of the number of SMs on the device.

Depending on other specific details in the code you have written, this refactor may or may not improve, or significantly change, the performance of your kernel. Therefore, as always, be sure to use `nvprof` so that you can quantitatively evaulate performance changes. Record the results with the rest of your findings thus far, based on the profiling output.

In [ ]:
```
!nvcc -arch=sm_70 -o sm-optimized-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nvprof ./sm-optimized-vector-add
```

Unified Memory Details

You have been allocting memory intended for use either by host or device code with `cudaMallocManaged` and up until now have enjoyed the benefits of this method - automatic memory migration, ease of programming - without diving into the details of how the **Unified Memory** (**UM**) allocated by `cudaMallocManaged` actual works. `nvprof` provides details about UM management in accelerated applications, and using this information, in conjunction with a more-detailed understanding of how UM works, provides additional opportunities to optimize accelerated applications.

The following slides present upcoming material visually, at a high level. Click through the slides before moving on to more detailed coverage of their topics in following sections.

In [2]:
```
%%HTML

<div align="center"><iframe src="https://docs.google.com/presentation/d/e/2PACX-1vS0-BCGiWUb82r1RH-4cSRmZjN2
```

Unified Memory Migration

When UM is allocated, the memory is not resident yet on either the host or the device. When either the host or device attempts to access the memory, a page fault (https://en.wikipedia.org/wiki/Page_fault) will occur, at which point the host or device will migrate the needed data in batches. Similarly, at any point when the CPU, or any GPU in the accelerated system, attempts to access memory not yet resident on it, page faults will occur and trigger its migration.

The ability to page fault and migrate memory on demand is tremendously helpful for ease of development in your accelerated applications. Additionally, when working with data that exhibits sparse access patterns, for example when it is impossible to know which data will be required to be worked on until the application actually runs, and for scenarios when data might be accessed by multiple GPU devices in an accelerated system with multiple GPUs, on-demand memory migration is remarkably beneficial.

There are times - for example when data needs are known prior to runtime, and large contiguous blocks of memory are required - when the overhead of page faulting and migrating data on demand incurs an overhead cost that would be better avoided.

Much of the remainder of this lab will be dedicated to understanding on-demand migration, and how to identify it in the profiler's output. With this knowledge you will be able to reduce the overhead of it in scenarios when it would be beneficial.

Exercise: Explore UM Page Faulting

`nvprof` provides output describing UM behavior for the profiled application. In this exercise, you will make several modifications to a simple application, and make use of `nvprof` 's Unified Memory output section after each change, to explore how UM data migration behaves.

`01-page-faults.cu` (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/01-page-faults.cu) contains a `hostFunction` and a `gpuKernel` , both which could be used to initialize the elements of a `2<<24` element vector with the number `1` . Curently neither the host function nor GPU kernel are being used.

For each of the 4 questions below, given what you have just learned about UM behavior, first hypothesize about what kind of page faulting should happen, then, edit `01-page-faults.cu` (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/01-page-faults.cu) to create a scenario, by using one or both of the 2 provided functions in the codebase, that will allow you to test your hypothesis.

In order to test your hypotheses, compile and profile your code using the code execution cells below. Be sure to record your hypotheses, as well as the results, obtained from `nvprof` output, specifically CPU and GPU page faults, for each of the 4 experiments you are conducting. There are links to solutions for each of the 4 experiments which you can refer to if you get stuck.

- What happens when unified memory is accessed only by the CPU? (solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/solutions/01-page-faults-solution-cpu-only.cu))

- What happens when unified memory is accessed only by the GPU? (solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/solutions/02-page-faults-solution-gpu-only.cu))
- What happens when unified memory is accessed first by the CPU then the GPU? (solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/solutions/03-page-faults-solution-cpu-then-gpu.cu))
- What happens when unified memory is accessed first by the GPU then the CPU? (solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/06-unified-memory-page-faults/solutions/04-page-faults-solution-gpu-then-cpu.cu))

In [ ]:
```
!nvcc -arch=sm_70 -o page-faults 06-unified-memory-page-faults/01-page-faults.cu -run
```

In [ ]:
```
!nvprof ./page-faults
```

Exercise: Revisit UM Behavior for Vector Add Program

Returning to the 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) program you have been working on throughout this lab, review the codebase in its current state, and hypothesize about what kinds of page faults you expect to occur. Look at the profiling output for your last refactor (either by scrolling up to find the output or by executing the code execution cell just below), observing the Unified Memory section of the profiler output. Can you explain the page faulting descriptions based on the contents of the code base?

In [ ]:
```
!nvprof ./sm-optimized-vector-add
```

Exercise: Initialize Vector in Kernel

When `nvprof` gives the amount of time that a kernel takes to execute, the host-to-device page faults and data migrations that occur during this kernel's execution are included in the displayed execution time.

With this in mind, refactor the `initWith` host function in your 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) program to instead be a CUDA kernel, initializing the allocated vector in parallel on the GPU. After successfully compiling and running the refactored application, but before profiling it, hypothesize about the following:

- How do you expect the refactor to affect UM page-fault behavior?

- How do you expect the refactor to affect the reported run time of `addVectorsInto` ?

Once again, record the results. Refer to the solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/07-init-in-kernel/solutions/01-vector-add-init-in-kernel-solution.cu) if you get stuck.

```
In [ ]: !nvcc -arch=sm_70 -o initialize-in-kernel 01-vector-add/01-vector-add.cu -run
```

```
In [ ]: !nvprof ./initialize-in-kernel
```

Asynchronous Memory Prefetching

A powerful technique to reduce the overhead of page faulting and on-demand memory migrations, both in host-to-device and device-to-host memory transfers, is called **asynchronous memory prefetching**. Using this technique allows programmers to asynchronously migrate unified memory (UM) to any CPU or GPU device in the system, in the background, prior to its use by application code. By doing this, GPU kernels and CPU function performance can be increased on account of reduced page fault and on-demand data migration overhead.

Prefetching also tends to migrate data in larger chunks, and therefore fewer trips, than on-demand migration. This makes it an excellent fit when data access needs are known before runtime, and when data access patterns are not sparse.

CUDA Makes asynchronously prefetching managed memory to either a GPU device or the CPU easy with its `cudaMemPrefetchAsync` function. Here is an example of using it to both prefetch data to the currently active GPU device, and then, to the CPU:

```
int deviceId;
cudaGetDevice(&deviceId);                                           // The ID of the currently active
 GPU device.

cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);        // Prefetch to GPU device.
cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId); // Prefetch to host. `cudaCpuDevic
eId` is a
                                                                    // built-in CUDA variable.
```

Exercise: Prefetch Memory

At this point in the lab, your 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) program should not only be launching a CUDA kernel to add 2 vectors into a third solution vector, all which are allocated with `cudaMallocManaged`, but should also initializing each of the 3 vectors in parallel in a CUDA kernel. If for some reason, your application does not do any of the above, please refer to the following reference application (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/08-prefetch/01-vector-add-prefetch.cu), and update your own codebase to reflect its current functionality.

Conduct 3 experiments using `cudaMemPrefetchAsync` inside of your 01-vector-add.cu (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/01-vector-add/01-vector-add.cu) application to understand its impact on page-faulting and memory migration.

- What happens when you prefetch one of the initialized vectors to the device?
- What happens when you prefetch two of the initialized vectors to the device?
- What happens when you prefetch all three of the initialized vectors to the device?

Hypothesize about UM behavior, page faulting specificially, as well as the impact on the reported run time of the initialization kernel, before each experiement, and then verify by running `nvprof`. Refer to the solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/08-prefetch/solutions/01-vector-add-prefetch-solution.cu) if you get stuck.

```
In [ ]: !nvcc -arch=sm_70 -o prefetch-to-gpu 01-vector-add/01-vector-add.cu -run
```

```
In [ ]: !nvprof ./prefetch-to-gpu
```

Exercise: Prefetch Memory Back to the CPU

Add additional prefetching back to the CPU for the function that verifies the correctness of the `addVectorInto` kernel. Again, hypothesize about the impact on UM before profiling in `nvprof` to confirm. Refer to the solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/08-prefetch/solutions/02-vector-add-prefetch-solution-cpu-also.cu) if you get stuck.

```
In [ ]: !nvcc -arch=sm_70 -o prefetch-to-cpu 01-vector-add/01-vector-add.cu -run
```

```
In [ ]: !nvprof ./prefetch-to-cpu
```

Summary

At this point in the lab, you are able to:

- Use the **NVIDIA Command Line Profiler** (**nvprof**) to profile accelerated application performance.
- Leverage an understanding of **Streaming Multiprocessors** to optimize execution configurations.
- Understand the behavior of **Unified Memory** with regard to page faulting and data migrations.
- Use **asynchronous memory prefetching** to reduce page faults and data migrations for increased performance.
- Employ an iterative development cycle to rapidly accelerate and deploy applications.

In order to consolidate your learning, and reinforce your ability to iteratively accelerate, optimize, and deploy applications, please proceed to this lab's final exercise. After completing it, for those of you with time and interest, please proceed to the *Advanced Content* section.

Final Exercise: Iteratively Optimize an Accelerated SAXPY Application

A basic accelerated SAXPY (https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_1) application has been provided for you here (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/09-saxpy/01-saxpy.cu). It currently contains a couple of bugs that you will need to find and fix before you can successfully compile, run, and then profile it with `nvprof` .

After fixing the bugs and profiling the application, record the runtime of the `saxpy` kernel and then work *iteratively* to optimize the application, using `nvprof` after each iteration to notice the effects of the code changes on kernel performance and UM behavior.

Utilize the techniques from this lab. To support your learning, utilize effortful retrieval (http://sites.gsu.edu/scholarlyteaching/effortful-retrieval/) whenever possible, rather than rushing to look up the specifics of techniques from earlier in the lesson.

Your end goal is to profile an accurate `saxpy` kernel, without modifying `N` , to run in under *50us*. Check out the solution (../../../../../edit/tasks/task1/task/02_AC_UM_NVPROF/09-saxpy/solutions/02-saxpy-solution.cu) if you get stuck, and feel free to compile and profile it if you wish.

```
In [ ]: !nvcc -arch=sm_70 -o saxpy 09-saxpy/01-saxpy.cu -run
```

```
In [ ]: !nvprof ./saxpy
```