COMPUTATIONAL PHYSICS: INCLUDES PARALLEL COMPUTING/PARALLEL PROGRAMMING

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Abstract. Everything about Computational Physics, including Parallel computing/ Parallel programming.

Part 1. Introduction

1. Parallel Computing

1.1. Udacity Intro to Parallel Programming : Lesson 1 - The GPU Programming Model.

Owens and Luebki pound fists at the end of this video. =)))) Intro to the class.

1.1.1. *Running CUDA locally.* Also, Intro to the class, in Lesson 1 - The GPU Programming Model, has links to documentation for running CUDA locally; in particular, for Linux: `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html`. That guide told me to go download the NVIDIA CUDA Toolkit, which is the https://developer.nvidia.com/cuda-downloads.

For *Fedora*, I chose Installer Type `runfile (local)`.

Afterwards, installation of CUDA on Fedora 23 workstation had been nontrivial. Go see either my github repository ML-grabbag (which will be updated) or my wordpress blog (which may not be upgraded frequently).

$P = VI = I^2 R$ heating.

1.1.2. *Definitions of Latency and throughput (or bandwidth).* cf. Building a Power Efficient Processor

Latency vs Bandwidth

latency [sec]. From the title "Latency vs. bandwidth", I'm thinking that throughput = bandwidth (???). throughput = job/time (of job).

Given total task, velocity v,
total task $/v =$ latency. throughput = latency/(jobs per total task).

Also, in Building a Power Efficient Processor. Owens recommends the article David Patterson, "Latency..."

cf. GPU from the Point of View of the Developer

$n_{\text{core}} \equiv$ number of cores
$n_{\text{vecop}} \equiv$ ($n_{\text{vecop}}-$wide axial vector operations/*core* core)
$n_{\text{thread}} \equiv$ threads/core (hyperthreading)

$$n_{\text{core}} \cdot n_{\text{vecop}} \cdot n_{\text{thread}} \text{ parallelism}$$

There were various websites that I looked up to try to find out the capabilities of my video card, but so far, I've only found these commands (and I'll print out the resulting output):

```
$ lspci -vnn | grep VGA -A 12
03:00.0 VGA compatible controller [0300]: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] [10de:17c8] (rev a1) (prog-if 00 [VG
        Subsystem: eVga.com. Corp. Device [3842:3994]
```

Date: 23 mai 2016.
Key words and phrases. Computational Physics, Parallel Computing, Parallel Programming.

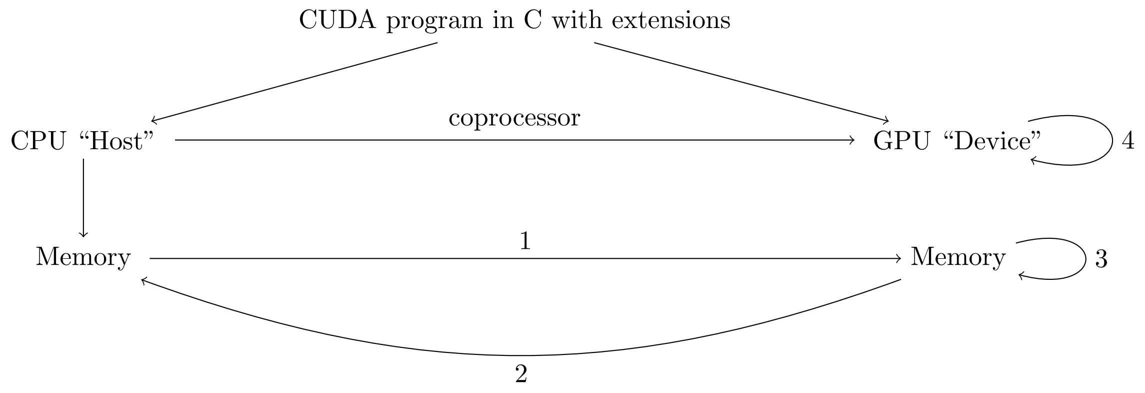
```
        Physical Slot: 4
        Flags: bus master, fast devsel, latency 0, IRQ 50
        Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
        Memory at e0000000 (64-bit, prefetchable) [size=256M]
        Memory at f0000000 (64-bit, prefetchable) [size=32M]
        I/O ports at e000 [size=128]
        [virtual] Expansion ROM at fb000000 [disabled] [size=512K]
        Capabilities: <access denied>
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia

$ lspci | grep VGA -E
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)

$ grep driver /var/log/Xorg.0.log
[    18.074] Kernel command line: BOOT_IMAGE=/vmlinuz-4.2.3-300.fc23.x86_64 root=/dev/mapper/fedora-root ro rd.lvm.lv=fedora/root rd.lvm.lv=fedora/swap rhgb quiet LANG=en_US.UTF-8 nouveau.modeset=0 rd.driver.blacklist=nouveau nomodeset gfxpaylo
[    18.087] (WW) Hotplugging is on, devices using drivers 'kbd', 'mouse' or 'vmmouse' will be disabled.
[    18.087]    X.Org XInput driver : 22.1
[    18.192] (II) Loading /usr/lib64/xorg/modules/drivers/nvidia_drv.so
[    19.088] (II) NVIDIA(GPU-0): Found DRM driver nvidia-drm (20150116)
[    19.102] (II) NVIDIA(0):     ACPI event daemon is available, the NVIDIA X driver will
[    19.174] (II) NVIDIA(0): [DRI2]   VDPAU driver: nvidia
[    19.284]    ABI class: X.Org XInput driver, version 22.1
...

$ lspci -k | grep -A 8 VGA
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia
03:00.1 Audio device: NVIDIA Corporation GM200 High Definition Audio (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: snd_hda_intel
        Kernel modules: snd_hda_intel
05:00.0 USB controller: VIA Technologies, Inc. VL805 USB 3.0 Host Controller (rev 01)
```

CUDA Program Diagram



CPU "host" is the boss (and issues commands) -Owen.

Coprocessor : CPU "host" → GPU "device"
Coprocessor : CPU process ↦ (co)-process out to GPU

With

1 data cpu → gpu
2 data gpu → cpu (initiated by cpu host)

1., 2., uses `cudaMemcpy`
3 allocate GPU memory: `cudaMalloc`
4 launch kernel on GPU

Remember that for 4., this launching of the kernel, while it's acting on GPU "device" onto itself, it's initiated by the boss, the CPU "host".

Hence, cf. Quiz: What Can GPU Do in CUDA, GPUs can respond to CPU request to receive and send Data CPU → GPU and Data GPU → CPU, respectively (1,2, respectively), and compute a kernel launched by the CPU (3).

A CUDA Program A typical GPU program

- `cudaMalloc` - CPU allocates storage on GPU
- `cudaMemcpy` - CPU copies input data from CPU → GPU
- *kernel launch* - CPU launches kernel(s) on GPU to process the data
- `cudaMemcpy` - CPU copies results back to CPU from GPU

Owens advises minimizing "communication" as much as possible (e.g. the `cudaMemcpy` between CPU and GPU), and do a lot of computation in the CPU and GPU, each separately.

Defining the GPU Computation

Owens circled this

BIG IDEA $\boxed{\text{This is Important}}$

Kernels look like serial programs

Write your program as if it will run on **one** thread

The GPU will run that program on **many** threads

Squaring A Number on the CPU

Note

(1) Only 1 thread of execution: ("thread" := one independent path of execution through the code) e.g. the `for` loop
(2) no explicit parallelism; it's serial code e.g. the `for` loop through 64 elements in an array

GPU Code A High Level View

CPU:

- Allocate Memory
- Copy Data to/from GPU
- Launch Kernel - species degree of parallelism

GPU:

- Express Out = In · In - says *nothing* about the degree of parallelism

Owens reiterates that in the GPU, everything looks serial, but it's only in the CPU that anything parallel is specified.

pseudocode: CPU code: square kernel <<< 64 >>> (outArray,inArray)

Squaring Numbers Using CUDA Part 3

From the example

```
// launch the kernel
square<<<1, ARRAY_SIZE>>>(d_out, d_in)
```

we're introduced to the "CUDA launch operator", initiating a kernel of 1 block of 64 elements (`ARRAY_SIZE` is 64) on the GPU. Remember that `d_` prefix (this is naming comvention) tells us it's on the device, the GPU, solely.

With CUDA launch operator ≡<<<>>>, then also looking at this explanation on `stackexchange` (so surely others are confused as well, of those who are learning this (cf. CUDA kernel launch parameters explained right?). From Eric's answer,

threads are grouped into blocks. all the threads will execute the invoked kernel function.

Certainly,

$$<<<>>>: (n_{\text{block}}, n_{\text{threads}}) \times \text{kernelfunctions} \mapsto \text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>> \in \text{End} : \text{Dat}_{\text{GPU}}$$
$$<<<>>>: \mathbb{N}^+ \times \mathbb{N}^+ \times \text{Mor}_{\text{GPU}} \to \text{EndDat}_{\text{GPU}}$$

where I propose that GPU can be modeled as a category containing objects Dat_{GPU}, the collection of all possible data inputs and outputs into the GPU, and Mor_{GPU}, the collection of all kernel functions that run (exclusively, and this *must* be the class, as reiterated by Prof. Owen) on the GPU.

Next,

$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: \text{din} \mapsto \text{dout} \qquad \text{(as given in the “square” example, and so I propose)}$$
$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: (\mathbb{N}^+)^{n_{\text{threads}}} \to (\mathbb{N}^+)^{n_{\text{threads}}}$$

But keep in mind that dout, din are pointers in the C program, pointers to the place in the memory.

`cudaMemcopy` is a functor category, s.t. e.g. $\text{Obj}_{\text{CudaMemcopy}} \ni \text{cudaMemcpyDevicetoHost}$ where

$$\text{cudaMemcopy}(-, -, n_{\text{thread}}, \text{cudaMemcpyDeviceToHost}) : \text{Memory}_{\text{GPU}} \to \text{Memory}_{\text{CPU}} \in \text{Hom}(\text{Memory}_{\text{GPU}}, \text{Memory}_{\text{CPU}})$$

Squaring Numbers Using CUDA 4

Note the C language construct *declaration specifier* - denotes that this is a kernel (for the GPU) and not CPU code. Pointers need to be allocated on the GPU (otherwise your program will crash spectacularly -Prof. Owen).

1.1.3. *What are C pointers?*

Is $\langle$ type $\rangle *$, a pointer, then a mapping from the category, namely the objects of types, to a mapping from the specified value type to a memory address?

e.g.

$$\langle\,\rangle * : \text{float} \mapsto \text{float} *$$
$$\text{float } * : \text{din} \mapsto \text{ some memory address}$$

and then we pass in mappings, not values, and so we're actually declaring a square *functor*.

What is `threadIdx`? What is it mathematically? Consider that $\exists$ 3 “modules”:

$$\text{threadIdx}.x$$
$$\text{threadIdx}.y$$
$$\text{threadIdx}.z$$

And then the line

```
int idx = threadIdx.x;
```

says that idx is an integer, “declares” it to be so, and then assigns idx to threadIdx.x which surely has to also have the same type, integer. So (perhaps)

$$idx \equiv \text{threadIdx}.x \in \mathbb{Z}$$

is the same thing.

Then suppose threadIdx $\subset$ FinSet, a subcategory of the category of all (possible) finite sets, s.t. threadIdx has 3 particular morphisms, $x, y, z \in \text{Mor} threadIdx$,

$$x : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$
$$y : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$
$$z : \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}}$$

Configuring the Kernel Launch Parameters Part 1

n_{blocks}, n_{threads} with $n_{\text{threads}} \geq 1024$ (this maximum constant is GPU dependent). You should pick the $(n_{\text{blocks}}, n_{\text{threads}})$ that makes sense for your problem, says Prof. Owen.

1.1.4. *Memory layout of blocks and threads.*

$\forall (n_{\text{blocks}}, n_{\text{threads}}) \in \mathbb{Z} \times \{1 \dots 1024\}$, $\{1 \dots n_{\text{block}} \times \{1 \dots n_{\text{threads}}\}$ is now an ordered index (with lexicographical ordering). This is just 1-dimensional (so possibly there's a 1-to-1 mapping to a finite subset of $\mathbb{Z}$).

I propose that “adding another dimension” or the 2-dimension, that Prof. Owen mentions is being able to do the Cartesian product, up to 3 Cartesian products, of the block-thread index.

Quiz: Configuring the Kernel Launch Parameters 2

Most general syntax:

Configuring the kernel launch

```
kernel<<<grid of blocks, block of threads >>>(...)

// for example

square<<<dim3(bx,by,bz), dim3(tx,ty,tz), shmem>>>(...)
```

where **dim3(tx,ty,tz)** is the grid of blocks $bx \cdot by \cdot bz$

{dim3}(tx,ty,tz) is the block of threads $tx \cdot ty \cdot tz$

shmem is the shared memory per block in bytes

Problem Set 1 “Also, the image is represented as an 1D array in the kernel, not a 2D array like I mentioned in the video.”

Here's part of that code for squaring numbers:

```
__global__ void square(float *d_out, float *d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f*f;
  }
```

1.1.5. *Grid of blocks, block of threads, thread that's indexed; (mathematical) structure of it all.*

Let

$$\text{grid} = \prod_{I=1}^{N} (\text{block})^{n_I^{\text{block}}}$$

where $N = 1, 2, 3$ (for CUDA) and by naming convention $\begin{aligned} I &= 1 \equiv x \\ I &= 2 \equiv y \\ I &= 3 \equiv z \end{aligned}$

Let's try to make it explicity (as others had difficulty understanding the grid, block, thread model, cf. colored image to greyscale image using CUDA parallel processing, Cuda gridDim and blockDim) through commutative diagrams and categories (from math):

$$\begin{tikzcd}
\prod_{I=1}^{N} \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[d, mapsto, "\text{dim3}"'] \\
\text{grid} \arrow[u, bend left, "\text{gridDim}"] & \ni \text{gridSize}(N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[u, mapsto, bend right, "(\text{gridDim}.x,\ \text{gridDim}.y,\ \text{gridDim}.z)"']
\end{tikzcd}$$

$$\begin{tikzcd}
\text{grid} \arrow[d, "\text{blockIdx}"] & \ni \texttt{d\_rgbaImage} \arrow[d, mapsto, "(\text{blockIdx}.x\text{,blockIdx}.y\text{,blockIdx}.z)"] \\
\prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1 \dots N_I^{\text{blocks}}\} & \ni (i^{\text{blocks}}, j^{\text{blocks}}, k^{\text{blocks}})
\end{tikzcd}$$

and then similar relations (i.e. arrows, i.e. relations) go for a block of threads:

$$\begin{tikzcd}
\prod_{I=1}^N \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[d, mapsto, "\text{dim3}"'] \\
\text{block} \arrow[u, bend left, "\text{blockDim}"] & \ni \text{blockSize}(N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[u, mapsto, bend right, "(\text{blockDim}.x,\ \text{blockDim}.y,\ \text{blockDim}.z)"']
\end{tikzcd}$$

$$\begin{tikzcd}
\text{block} \arrow[d, "\text{threadIdx}"] & \ni \text{block} \arrow[d, mapsto, "(\text{threadIdx}.x\text{,threadIdx}.y\text{,threadIdx}.z)"] \\
\prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1 \dots N_I^{\text{threads}}\} & \ni (i^{\text{threads}}, j^{\text{threads}}, k^{\text{threads}})
\end{tikzcd}$$

gridsize help assignment 1 Pp explains how threads per block is variable, and remember how Owens said Luebki says that a GPU doesn't get up for more than a 1000 threads per block.

1.1.6. *Generalizing the model of an image.* Consider vector space V, e.g. $\dim V = 4$, vector space V over field $\mathbb{K}$, so $V = \mathbb{K}^{\dim V}$.

Each pixel represented by $\forall\, v \in V$.

Consider an image, or space, M. $\dim M = 2$ (image), $\dim M = 3$. Consider a local chart (that happens to be global in our case):

$$\varphi : M \to \mathbb{Z}^{\dim M} \supset \{1 \dots N_1\} \times \{1 \dots N_2\} \times \dots \times \{1 \dots N_{\dim M}\}$$
$$\varphi : x \mapsto (x^1(x), x^2(x), \dots, x^{\dim M}(x))$$

$$\begin{tikzcd}
E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & M \times V \arrow[dl] \\
M &
\end{tikzcd}
\qquad
\begin{tikzcd}
E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & \text{grid} \times\ \text{block of threads} \arrow[dl] \\
\text{grid} &
\end{tikzcd}$$

Consider a "coarsing" of underlying M:

$$\begin{tikzcd}
M \times V \arrow[r, "\text{proj}"] \arrow[d, "\pi"'] & \text{proj}(M) \times \text{proj}(V) \arrow[d, "\text{proj}(\pi)"] \\
M = \{1 \dots N_1\} \times \{1 \dots N_2\} \times \dots \times \{1 \dots N_{\dim M}\} \arrow[r, "\text{proj}"] & \text{proj}(M) = \{1 \dots \frac{N_1}{N_1^{\text{threads}}}\} \times \{1 \dots \frac{N_2}{N_2^{\text{threads}}}\} \times \dots \times \{1 \dots \frac{N_{\dim M}}{N_{\dim M}^{\text{threads}}}\}
\end{tikzcd}$$

e.g. $N_1^{\text{thread}} = 12$

$N_2^{\text{thread}} = 12$

Just note that in terms of syntax, you have the "block" model, in which you allocate blocks along each dimension. So in

$$const\ dim3\ blockSize(n_x^b, n_y^b, n_z^b)$$
$$const\ dim3\ gridSize(n_x^{\text{gr}}, n_y^{\text{gr}}, n_z^{\text{gr}})$$

Then the condition is $n_x^b/\dim V, n_y^b/\dim V, n_z^b/\dim V \in \mathbb{Z}$ (condition), $\qquad (n_x^{\text{gr}} - 1)/\dim V, n_y^{\text{gr}}/\dim V, n_z^{\text{gr}}/\dim V \in \mathbb{Z}$

Transpose Part 1

Now

$$\text{Mat}_{\mathbb{F}}(n,n) \xrightarrow{T} \text{Mat}_{\mathbb{F}}(n,n)$$
$$A \mapsto A^T \text{ s.t. } (A^T)_{ij} = A_{ji}$$
$$\text{Mat}_{\mathbb{F}} \xrightarrow{T} \mathbb{F}^{n^2}$$
$$A_{ij} \mapsto A_{ij} = A_{in+j}$$

$$\begin{tikzcd}
\text{Mat}_{\mathbb{F}}(n,n) \arrow[r] \arrow[d, "T"'] & \mathbb{F}^{n^2} \arrow[d, "T"] \\
\text{Mat}_{\mathbb{F}}(n,n) \arrow[r] & \mathbb{F}^{n^2}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{ij} \arrow[r, mapsto] \arrow[d, mapsto, "T"'] & A_{in+j} \arrow[d, mapsto, "T"] \\
(A^T)_{ij} = A_{ji} \arrow[r, mapsto] & A_{jn+i}
\end{tikzcd}$$

Transpose Part 2

Possibly, transpose is a functor.

Consider struct as a category. In this special case, Objstruct = {arrays} (a struct of arrays). Now this struct already has a hash table for indexing upon declaration (i.e. "creation"): so this category struct will need to be equipped with a "diagram" from the category of indices J to struct: $J \to$ struct.

So possibly

$$\text{struct} \xrightarrow{T} \qquad \text{array}$$
$$\text{ObjStruct} = \{\text{ arrays }\} \xrightarrow{T} \text{ Objarray} = \{\text{ struct }\}$$
$$J \to \text{ struct } \xrightarrow{T} \qquad J \to \text{ array}$$

Quiz: What Kind Of Communication Pattern This quiz made a few points that clarified the characteristics of these so-called communication patterns (amongst the memory?)

- map is bijective, and map : Idx → Idx
- gather - not necessarily surjective
- scatter - not necessarily surjective
- stencil - surjective
- transpose (see before)

Parallel Communication Patterns Recap

- map - bijective
- transpose - bijective
- gather - not necessarily surjective, and is many-to-one (by def.)
- scatter - one-to-many (by def.) and is not necessarily surjective
- stencil - several-to-one (not injective, by definition), and is surjective
- reduce - all-to-one
- scan/sort - all-to-all

Programmer View of the GPU

thread blocks: group of threads that cooperate to solve a (sub)problem

Thread Blocks And GPU Hardware

CUDA GPU is a bunch of SMs:
Streaming Multiprocessors (SM)s
SMs have a bunch of simple processors and memory.

Dr. Luebki:

> Let me say that again because it's really important
> GPU is responsible for allocating blocks to SMs

Programmer only gives GPU a pile of blocks.

Quiz: What Can The Programmer Specify

I myself thought this was a revelation and was not intuitive at first:

Given a single kernel that's launched on many thread blocks include X, Y, the programmer cannot specify the sequence the blocks, e.g. block X, block Y, run (same time, or run one after the other), and which SM the block will run on (GPU does all this).

Quiz: A Thread Block Programming Example

Open up `hello blockIdx.cu` in Lesson 2 Code Snippets (I got the repository from github, repo name is cs344).

At first, I thought you can do a single file compile and run in Eclipse without creating a new project. No. cf. Eclipse creating projects every time to run a single file?.

I ended up creating a new CUDA C/C++ project from File -¿ New project, and then chose project type Executable, Empty Project, making sure to include Toolchain CUDA Toolkit (my version is 7.5), and chose an arbitrary project name (I chose cs344single). Then, as suggested by Kenny Nguyen, I dragged and dropped files into the folder, from my file directory program.

I ran the program with the "Play" triangle button, clicking on the green triangle button, and it ran as expected. I also turned off Build Automatically by deselecting the option (no checkmark).

GPU Memory Model

$$\begin{tikzcd} \text{thread} & \text{local memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

Then consider threadblock $\equiv$ thread block

$$\text{Objthreadblock} \supset \{ \text{ threads } \}$$
$$\text{FinSet} \xrightarrow{\text{threadIdx}} \text{thread} \ \in \text{Morthreadblock}$$

$$\begin{tikzcd} \text{threadblock} & \text{shared memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

$\forall$ thread,

$$\begin{tikzcd} \text{thread} & \text{global memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"] \end{tikzcd}$$

Synchronization - Barrier

Quiz: The Need For Barriers

3 barriers were needed (wasn't obvious to me at first). All threads need to finish the write, or initialization, so it'll need a barrier.

While

```
array[idx] = array[idx+1];
```

is 1 line, it'll actually need 2 barriers; first read. Then write.

So *actually* we'll need to *rewrite* this code:

```
int temp = array[idx+1];
__syncthreads();
array[idx] = temp;
__syncthreads();
```

kernels have implicit barrier for each.

Writing Efficient Programs

(1) Maximize *arithmetic intensity* arithmetic intensity $:= \frac{\text{math}}{\text{memory}}$

video: Minimize Time Spent On Memory

local memory is fastest; global memory is slower

$$\text{local} > \ \text{shared} >> \text{global} >> \text{CPU}$$

kernel we know (in the code) is tagged with `__global__`

quiz: A Quiz on Coalescing Memory Access

Work it out as Dr. Luebki did to figure out if it's coalesced memory access or not.

Atomic Memory Operations

Atomic Memory Operations
atomicadd atomicmin atomicXOR atomicCAS Compare And Swap

2. Pointers in C; Pointers in C categorified (interpreted in Category Theory)

Suppose $v \in \text{ObjData}$, category of data **Data**,
e.g. $v \in \text{Int} \in \text{ObjType}$, category of types Type.

$$\text{Data} \xrightarrow{\&} \text{Memory}$$
$$v \overset{\&}{\mapsto} \&v$$

with address $\&v \in \text{Memory}$.

With
assignment $pv = \&v$,

$$pv \in \text{Objpointer}, \ \text{ category of pointers, pointer}$$
$$pv \in \text{Memory} \qquad (\text{i.e. not } pv \in \text{Dat, i.e. } pv \notin \text{Dat})$$

$$\text{pointer } \ni pv \overset{*}{\mapsto} *pv \in \text{Dat}$$

$$\begin{tikzcd} v \arrow[r, mapsto, "\&"] & \&v \arrow[d, mapsto, "="] \\ *pv \arrow[u, mapsto, "=="] & pv \arrow[l, mapsto, "*"] \end{tikzcd} \qquad \begin{tikzcd} \text{Data} \arrow[r, "\&"] & \text{Memory} \arrow[d, "="] \\ \text{Data} \arrow[u, "=="] & \text{pointer} \arrow[l, "*"] \end{tikzcd}$$

Examples. Consider `passfunction.c` in Fitzpatrick [5].

Consider the type `double`, `double` $\in$ ObjTypes.

fun1, fun2 $\in$ MorTypes namely
fun1, fun2 $\in \text{Hom}(\text{double}, \text{double}) \equiv \text{Hom}_{\text{Types}}(\text{double}, \text{double})$

Recall that

$$\text{pointer} \xrightarrow{*} \text{Dat}$$

$$\text{pointer} \xrightarrow{\&} \text{Memory}$$

$*, \&$ are functors with domain on the category pointer.

Pointers to functions is the "extension" of functor $*$ to the codomain of MorTypes:

$$\begin{aligned} &\text{pointer} && \xrightarrow{*} \text{MorTypes} \\ &\text{fun1} && \overset{*}{\mapsto} *\text{fun1} \in \text{Hom}_{\text{Types}}(\text{double}, \text{double}) \end{aligned}$$

$$\begin{tikzcd} \text{double} \arrow[r, "\&"] & \text{Memory} \arrow[d, "\cong"] \\ \text{double} \arrow[d, "\text{cube}"'] & \text{pointer} \arrow[l, "*"] \arrow[dl, "*"] \\ \text{double} & \end{tikzcd} \qquad \begin{tikzcd} \text{res1} \arrow[r, mapsto, "\&"] & \&\text{res1} \arrow[d, mapsto, "\cong"] \\ *\text{res1} \arrow[d, mapsto, "\text{cube}"'] & \text{res1} \arrow[l, mapsto, "*"] \arrow[dl, mapsto, "*"] \\ *\text{res1} = y^3 & \end{tikzcd}$$

It's unclear to me how `void cube` can be represented in terms of category theory, as surely it cannot be represented as a mapping (it acts upon a functor, namely the $*$ functor for pointers). It doesn't return a value, and so one cannot be confident to say there's explicitly a domain and codomain, or range for that matter.

But what is going on is that

$$\text{pointer}\,,\ \text{double}\,,\ \text{pointer} \xrightarrow{\text{cube}} \text{pointer}\,,\ \text{pointer}$$

$$\text{fun1}, x, \text{res1} \overset{\text{cube}}{\mapsto} \text{fun1}, \text{res1}$$

s.t. $*\text{res1} = y^3 = (*\text{fun1}(x))^3$

So I'll speculate that in this case, `cube` is a functor, and in particular, is acting on $*$, the so-called deferencing operator:

$$\begin{matrix} \text{pointer} \xrightarrow{*} \text{float} \in \text{Data} & & \text{pointer} \xrightarrow{\text{cube}(*)} \text{float} \in \text{Data} \\ \text{res1} \overset{*}{\mapsto} *\text{res1} & \xrightarrow{\text{cube}} & \text{res1} \overset{\text{cube}(*)}{\mapsto} \text{cube}(*\text{res1}) = y^3 \end{matrix}$$

cf. Arrays, from Fitzpatrick [5]

$$\text{Types} \xrightarrow{\text{declaration}} \text{arrays}$$

If $x \in \text{Objarrays}$,

$$\&x[0] \in \text{Memory} \xrightarrow{==} x \in \text{ pointer (to 1st element of array)}$$

cf. Section 2.13 Character Strings from Fitzpatrick [5]

```
char word[20] = ‘‘four’’
char *word = ‘‘four’’
```

cf. C++ extensions for C according to Fitzpatrick [5]

- simplified syntax to pass by reference pointers into functions
- inline functions
- variable size arrays

```
int n;
double x[n];
```

- complex number class

Part 2. C++ and Computational Physics

cf. 2.1.1 Scientific hello world from Hjorth-Jensen (2015) [6]
in C,

```
int main (int argc, char* argv[])
```

`argc` stands for number of command-line arguments
`argv` is vector of strings containing the command-line arguments with
`argv[0]` containing name of program
`argv[1] , argv[2], ...` are command-line args, i.e. the number of lines of input to the program

"To obtain an executable file for a C++ program" (i.e. compile (???)),

```
gcc -c -Wall myprogram.c
gcc -o myprogram myprogram.o
```

`-Wall` means warning is issued in case of non-standard language
`-c` means compilation only
`-o` links produced object file `myprogram.o` and produces executable `myprogram`

```
# General makefile for c - choose PROG =  name of given program

# Here we define compiler option, libraries and the target
CC= c++ -Wall
PROG= myprogram

# Here we make the executable file
${PROG} :            ${PROG}.o
                     ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

#{PROG}.o :           ${PROG}.cpp
                     ${CC} -c ${PROG}.cpp
```

Here's what worked for me:

```
CC= g++ -Wall
PROG= program1

# Here we make the executable file
${PROG} :            ${PROG}.o
        ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :          ${PROG}.cpp
```

```
        ${CC} -c ${PROG}.cpp

# EY : 20160602 notice the different suffixes , and we see the pattern for the syntax

# (note: the <tab> in the command line is necessary formake towork)
# target: dependency1 dependency2 ...
#         <tab> command
```

cf. 2.3.2 Machine numbers of Hjorth-Jensen (2015) [6]
cf. 2.5.2 Pointers and arrays in C++ of Hjorth-Jensen (2015) [6]
Initialization (diagram):

$$\begin{tikzcd}
\&\text{var} = \texttt{0x7ffc97efbd8c} \arrow[r, mapsto, "="] & \text{pointer} = \&\text{var} = \texttt{0x7ffc97efbd8c}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Memory} \arrow[r, "="] & \text{pointer}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{(memory) addresses} \arrow[r, "="] & \text{Obj(pointer)}
\end{tikzcd}$$

Referencing and deferencing operations on pointers to variables

$$\begin{tikzcd}
\text{var} \arrow[r, mapsto, "="] & \&\text{var} = \texttt{0x7ffc97egbd8c} \\
\text{var}, 421 \arrow[u, mapsto] \arrow[r, mapsto, "=="] \arrow[d, mapsto] & \text{True} \\
421 &
\end{tikzcd}$$

$$\begin{tikzcd}
\text{pointer} = \texttt{0x7ffc97egbd8c} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & 421 \arrow[r, mapsto, "\text{typedef}"] & \text{int} \\
\&\text{pointer} = \texttt{0x7ffc97egbd8c} & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "="] & \mathbf{Memory} \\
\mathbf{Types} \otimes \mathbf{Dat} \arrow[u] \arrow[r, "=="] \arrow[d] & \text{Boolean} = \{\text{True}, \text{False}\} \\
\mathbf{Dat} &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat} \arrow[r, "\text{typedef}"] & \mathbf{Types} \\
\mathbf{Memory} & &
\end{tikzcd}$$

2.1. **Numerical differentiation and interpolation (in C++).** cf. Chapter 3 "Numerical differentiation and interpolation" of Hjorth-Jensen (2015) [6].

This is how I understand it.

Consider the Taylor expansion for $f(x) \in C^\infty(\mathbb{R})$:

$$f(x) = f(x_0) + \sum_{j=1}^\infty \frac{f^{(j)}(x_0)}{j!} h^j$$

For $x = x_0 \pm h$,

$$f(x) = f(x_0 \pm h) = f(x_0) + \sum_{j=1}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} h^{2j} \pm \sum_{j=1}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} h^{2j-1}$$

Then

$$\begin{aligned}
f(x_0 + 2^k h) - f(x_0 - 2^k h) &= 2 \sum_{j=1}^\infty \frac{f^{(2j-1)}}{(2j-1)!}(x_0) 2^{k(2j-1)} h^{2j-1} = \\
&= 2 \left[f^{(1)}(x_0) 2^k h + \sum_{j=2}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} 2^{k(2j-1)} h^{2j-1} \right] = \\
&= 2 \left[f^{(1)}(x_0) 2^k h + \frac{f^{(3)}(x_0)}{3!} 2^{k(3)} h^3 + \sum_{j=3}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} 2^{k(2j-1)} h^{2j-1} \right]
\end{aligned}$$

So for $k = 1$,

$$f(x_0 + h) - f(x_0 - h) = 2 \left[f^{(1)}(x_0) h + \sum_{j=1}^\infty \frac{f^{(2j+1)}(x_0)}{(2j+1)!} h^{2j+1} \right]$$

Now

$$\begin{gathered}
f(x_0 + 2^k h) + f(x_0 - 2^k h) - 2f(x_0) = \\
= 2 \sum_{j=1}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} = \\
= 2 \left[\frac{f^{(2)}(x_0)}{2} 2^{2k} h^2 + \sum_{j=2}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} \right] = \\
= 2 \left[\frac{f^{(2)}(x_0)}{2} 2^{2k} h^2 + \frac{f^{(4)}(x_0)}{4!} 2^{4k} h^4 + \sum_{j=3}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} \right]
\end{gathered}$$

Thus for the case of $k = 1$,

$$f(x_0 + h) + f(x_0 - h) - 2f(x_0) = f^{(2)}(x_0) h^2 + 2 \sum_{j=2}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} h^{2j}$$

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h} = f^{(1)}(x_0) + \sum_{j=1}^\infty \frac{f^{(2j+1)}(x_0)}{(2j+1)!} h^{2j}$$

$$\frac{f(x_0 + h) + f(x_0 - h) - 2f(x_0)}{h^2} = f^{(2)}(x_0) + 2 \sum_{j=2}^\infty \frac{f^{(2(j+1))}(x_0)}{(2(j+1))!} h^{2j}$$

A pattern now emerges on how to include more calculations at points $x_0, x_0 \pm 2^k h$ so to obtain better accuracy $O(h^l)$. For instance,

Given 5 pts. $\{x_0, x_0 \pm h, x_0 \pm 2h\}$,

$$\begin{gathered}
f(x_0 + 2h) - f(x_0 - 2h) = 2[f^{(1)}(x_0) 2^1 h + \frac{f^{(3)}(x_0)}{3!} 2^3 h^3 + O(h^5)] \\
f(x_0 + h) - f(x_0 - h) = 2[f^{(1)}(x_0) h + \frac{f^{(3)}(x_0)}{3!} h^3 + O(h^5)] \\
\implies f'(x_0) = \frac{f(x_0 - 2h) - 8f(x_0 - h) + 8f(x_0 + h) - f(x_0 + 2h)}{12h} + O(h^4)
\end{gathered}$$

Hjorth-Jensen (2015) [6] argues, on pp. 46-47, that the additional evaluations are time consuming, to obtain further accuracy, so it's a balance.

To summarize, for $O(h^2)$ accuracy,

$$\begin{aligned} \frac{f(x_0+h)-f(x_0-h)}{2h} &= f^{(1)}(x_0) + \sum_{j=1}^{\infty} \frac{f^{(2j+1)}(x_0)}{(2j+1)!}h^{2j} && O(h^2) \\ \frac{f(x_0+h)+f(x_0-h)-2f(x_0)}{h^2} &= f^{(2)}(x_0) + 2\sum_{j=1}^{\infty} \frac{f^{(2j+2)}(x_0)}{(2j+2)!}h^{2j} && O(h^2) \end{aligned}$$

3. Interpolation

cf. 3.2 Numerical Interpolation and Extrapolation of Hjorth-Jensen (2015) [6]

Given $N+1$ pts. $\begin{aligned} y_0 &= f(x_0) \\ y_1 &= f(x_1) \\ &\vdots \\ y_N &= f(x_N) \end{aligned}$, x_i's distinct (none of x_i values equal)

We want a polynomial of degree n s.t. $p(x) \in \mathbb{R}[x]$

$$p(x_i) = f(x_i) = y_i \qquad i = 0, 1 \dots N$$

$$p(x) = a_0 + a_1(x-x_0) + \dots + a_i \prod_{j=0}^{i-1}(x-x_j) + \dots + a_N(x-x_0)\dots(x-x_{N-1}) = a_0 + \sum_{i=1}^N a_i \prod_{j=0}^{i-1}(x-x_j)$$

$$\begin{aligned} & a_0 = f(x_0) \\ & a_0 + a_1(x_1 - x_0) = f(x_1) \\ & \vdots \\ & a_0 + \sum_{i=1}^k a_i \prod_{j=0}^{i-1}(x_k - x_j) = f(x_k) \end{aligned}$$

Hjorth-Jensen (2015) [6] mentions this Lagrange interpolation formula (I haven't found a good proof for it).

$$\boxed{p_N(x) = \sum_{i=0}^N \prod_{k\neq i} \frac{x-x_k}{x_i - x_k} y_i} \tag{1}$$

4. Classes (C++)

cf. C++ Operator Overloading in expression

Take a look at this link: C++ Operator Overloading in expression. This point isn't emphasized enough, as in Hjorth-Jensen (2015) [6]. This makes doing something like

$$d = a*c + d/b$$

work the way we expect. Kudos to user fredoverflow for his answer:

"The expression `(e_x*u_c)` is an rvalue, and references to non-const won't bind to rvalues.

Also, member functions should be marked `const` as well."

4.1. What are lvalues and rvalues in C and C++? C++ Rvalue References Explained

Original definition of *lvalues* and *rvalues* from *C*:

lvalue - expression e that may appear on the left or on the right hand side of an assignment

rvalue - expression that can only appear on right hand side of assignment $=$.

Examples:

```
int a = 42;
int b = 43;

// a and b are both l-values
a = b; // ok
b = a; // ok
a = a * b; // ok

// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

In *C++*, this is still useful as a first, intuitive approach, but

lvalue - expression that refers to a memory location and allows us to take the address of that memory location via the & operator.

rvalue - expression that's not a lvalue

So & reference *functor* can't act on rvalue's.

5. Numerical Integration

5.0.1. *Trapezoid rule (or trapezoidal rule).* See Integrate.ipynb.

From there, consider integration on $[a,b]$, considering $h := \frac{b-a}{N}$, and $N+1$ (grid) points, $\{a, a+h, a+2h, \dots, a+jh, \dots, a+Nh = b\}_{j=0\dots N}$.

Then $\frac{N}{2}$ pts. are our "x_0"; x_0's $= \{a+h, a+3h, \dots, a+(2j-1)h, \dots, a+\left(\frac{2N}{2}-1\right)h\}_{j=1\dots\frac{N}{2}}$.

Notice how we really need to care about if N is even or not. If N is not even, then we'd have to deal with the integration at the integration limits and choosing what to do.

Then

$$\begin{gathered} \int_a^b f(x)dx = \sum_{j=1}^{N/2} \int_{a+(2j-1)h-h}^{a+(2j-1)h+h} f(x)dx = \sum_{j=1}^{N/2} \frac{h}{2}(2f(a+(2j-1)h) + f(a+2(j-1)h) + f(a+2jh)) = \\ = h(f(a)/2 + f(a+h) + \dots + f(b-h) + \frac{f(b)}{2}) = h\left(\frac{f(a)}{2} + \sum_{j=1}^{N-1} f(a+jh) + \frac{f(b)}{2} \right) \end{gathered}$$

5.0.2. *Midpoint method or rectangle method.* .

Let $h := \frac{b-a}{N}$ be the step size. The grid is as follows:

$$\{a, a+h, \dots, a+jh, \dots, a+Nh = b\}_{j=0\dots N}$$

The desired midpoint values are at the following N points:

$$\{a + \frac{h}{2}, a + \frac{3}{2}h, \dots, a + \frac{(2j-1)h}{2}, \dots, a + \left(N - \frac{1}{2}\right)h\}_{j=1\dots N}$$

and so

$$\int_a^b f(x)dx \approx \sum_{j=1}^N f(x_j)h = \sum_{j=1}^N f\left(a + \frac{(2j-1)h}{2}\right)h \tag{2}$$

5.0.3. *Simpson rule.* The idea is to take the next "order" in the Lagrange interpolation formula, the second-order polynomial, and then we can rederive Simpson's rule. The algebra is worked out in Integrate.ipynb.

From there, then we can obtain Simpson's rule,

$$\int_a^b f(x)dx = \sum_{j=1}^{N/2} \int_{a+2(j-1)h}^{a+2jh} f(x)dx = \sum_{j=1}^{N/2} \frac{h}{3}(4f(a+(2j-1)h) + f(a+2(j-1)h) + f(a+2jh)) =$$

$$= \frac{h}{3}\left[f(a) + f(b) + \sum_{j=1}^{N/2} 4f(a+(2j-1)h) + 2\sum_{j=1}^{N/2-1} f(a+2jh) \right]$$

5.1. **Gaussian Quadrature.** cf. Hjorth-Jensen (2015) [6], Section 5.3 Gaussian Quadrature, Chapter 5 Numerical Integration

6. Call by reference - Call by Value, Call by reference (in C and in C++)

cf. pp. 58, 2.10 Pointers Ch. 2 Scientific Programming in C, Fitzpatrick [5] `printfact3.c`, printfact3.c
pass pointer, pass by reference, call by pointer, call by reference
In C:

- *function prototype* -

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \mathbf{Types} \\ \mathbf{Types} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, mapsto, "\text{function}"] \arrow[d, mapsto, "*"] & \text{void} \\ \text{double} & \end{tikzcd}$$

$\Longrightarrow$

```
void factorial(double *)
```

where for factorial, it's just your choice of name for *function*.

- *function definition* -

$$\begin{tikzcd} \mathbf{pointers} \arrow[r, "\text{function}"] \arrow[d, "*"] & \mathbf{Types} \\ \mathbf{Types} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \text{void} \\ \text{double} & \end{tikzcd} \qquad \begin{tikzcd} \text{fact} \arrow[r, mapsto, "\text{function}"] \arrow[d, mapsto, "*"] & \text{void} \\ *\text{fact} & \end{tikzcd}$$

$\Longrightarrow$

```
void function(double *fact) { ... }
```

Inside the function definition,

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types} \\ \mathbf{Memory} & & \end{tikzcd} \qquad \begin{tikzcd} \text{fact} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & *\text{fact} \arrow[r, mapsto, "\text{typedef}"] & \text{double} \\ \&\text{fact} & & \end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
*fact = 1
*fact *= (double) n
```

and so notice that from `*fact = 1`, `*fact` is a lvalue.

– *function procedure*

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat}_{\text{lvalues}} \arrow[loop right, "\text{function procedure}"] \\ \mathbf{Memory} & \end{tikzcd} \qquad \begin{tikzcd} \text{fact} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & *\text{fact} \arrow[loop right, "\text{function procedure}"] \\ \&\text{fact} & \end{tikzcd}$$

$\Longrightarrow$

```
*fact *= (double) n
```

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd} \mathbf{Types} \arrow[loop left, "\text{function procedure}"] \arrow[r, "\&"] & \mathbf{Memory} \arrow[d, "\cong"] & \\ & \mathbf{pointers} \arrow[r, "\text{function}"] & \mathbf{Types} \end{tikzcd}$$

$$\begin{tikzcd} \text{double} \arrow[loop left, "\text{function procedure}"] \arrow[r, "\&"] & \text{Memory}(\text{Obj}\mathit{Memory}) \arrow[d, "\cong"] & \\ & \text{pointer} \arrow[r, "\text{function}"] & \text{void} \end{tikzcd}$$

$$\begin{tikzcd} \text{fact} \arrow[loop left, "\text{function procedure}"] \arrow[r, mapsto, "\&"] & \&\text{fact} \arrow[d, mapsto, "\cong"] & \\ & \&\text{fact} \arrow[r, mapsto, "\text{function}"] & \text{function}(\&\ \text{fact}) \end{tikzcd}$$

where, again simply note the notation, that we're using *function* and *factorial*, *fact* for *nameofpointer*, interchangeably: see printfact3.c for the example I'm referring to.

Again, *in C*, consider *a pointer to a function* passed to another function as an argument. Take a look at passfunction.c simultaneously.

- *function prototype* -

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\ \text{Mor}_{\mathbf{Types}} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{void} \\ \text{Mor}_{\mathbf{Types}}(\text{double}, \text{double}) & \end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction(double (*)(double))
```

We could further generalize this syntax, simply for syntax and notation sake, as such:

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\ \text{Mor}_{\mathbf{Types}} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{data-type} \\ \text{Mor}_{\mathbf{Types}}(\text{typei}, \text{typef}) & \end{tikzcd}$$

$\Longrightarrow$

```
data−type hostfunction(typef (*)(typei))
```

For practice, consider more than 1 argument in our function, and the other argument, for practice, is a pointer, we're "passing by reference."

$$\begin{tikzcd}
& \mathbf{pointers}\times\mathbf{pointers} \arrow[dl, "\text{pr}_1"'] \arrow[dr, "\text{pr}_2"] \arrow[rr, "\text{hostfunction}"] & & \mathbf{Types} \\
\mathbf{pointers} \arrow[d, "*"] & & \mathbf{pointers} \arrow[d, "*"] & \\
\text{Mor}_{\mathbf{Types}} & & \mathbf{Types} &
\end{tikzcd}$$

$$\begin{tikzcd}
& \textit{pointer}\times\textit{pointer} \arrow[dl, mapsto, "\text{pr}_1"'] \arrow[dr, mapsto, "\text{pr}_2"] \arrow[rr, mapsto, "\text{hostfunction}"] & & \text{void} \\
\textit{pointer} \arrow[d, mapsto, "*"] & & \textit{pointers} \arrow[d, mapsto, "*"] & \\
\text{Mor}_{\mathbf{Types}}(\text{double},\text{double}) & & \text{double} &
\end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction( double (*)(double), double *)
```

- *function definition*

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \text{void} \\
\text{Mor}_{\mathbf{Types}}(\text{double},\text{double}) &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{fun} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{void} \\
*\text{fun} &
\end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction(double (*fun)(double)) { ... }
```

- *Inside* the function definition,

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "*\text{fun}"] & \mathbf{Types} \arrow[r, "="] & \mathbf{Types} \\
\text{double} \arrow[r, "*\text{fun}"] & \text{double} \arrow[r, "="] & \text{double} \\
x \arrow[r, mapsto, "*\text{fun}"] & (*\text{fun})(x) \arrow[r, mapsto, "="] & y = (*\text{fun})(x)
\end{tikzcd}$$

$\Longrightarrow$

```
y = (*fun)(x)
```

- "Using" the function - the *actual* syntax for "passing" a function into a function is interesting (peculiar?): you only need the *name* of the function.

  Let's quickly recall how a function is prototyped, "declared" (or, i.e., defined), and used:
  - *function prototype* -

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "\text{fun1}"] & \mathbf{Types} \\
\text{double} \arrow[r, "\text{fun1}"] & \text{double}
\end{tikzcd}$$

$\Longrightarrow$

```
double fun1(double)
```

  - *function definition* -

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "\text{fun1}"] & \mathbf{Types} \\
\text{double} \arrow[r, "\text{fun1}"] & \text{double} \\
z \arrow[r, mapsto, "\text{fun1}"] & 3.0z*z-z(=3z^2-z)
\end{tikzcd}$$

$\Longrightarrow$

```
double fun1(double z) { ... }
```

  - Using function - `fun1(z)`

and so

$$\text{fun1}\in\text{Mor}_{\mathbf{Types}}(\text{double},\text{double})$$

And so again, it's interesting in terms of syntax that all you need is the *name* of the function to pass into the arguments of the "host function" when using the host function:

$$\begin{tikzcd}
\text{Mor}_{\mathbf{Types}} \arrow[r, "\text{hostfunction}"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}}(\text{double},\text{double}) \arrow[r, mapsto, "\text{hostfunction}"] & \text{void} \\
\text{fun1} \arrow[r, mapsto, "\text{hostfunction}"] & \text{hostfunction}(\text{fun1})
\end{tikzcd}$$

$\Longrightarrow$

```
hostfunction(fun1)
```

6.0.1. *C++ extensions, or how C++ pass by reference (pass a pointer to argument) vs. C.* Recall how C passes by reference, and look at Fitzpatrick [5], pp. 83-84 for the `square` function:

- *function prototype* -

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "\text{square}"] \arrow[d, "*"] & \mathbf{Types} \\
\mathbf{Types} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{pointer} \arrow[r, mapsto, "\text{square}"] \arrow[d, mapsto, "*"] & \text{void} \\
\text{double} &
\end{tikzcd}$$

$\Longrightarrow$

```
void square(double *)
```

- *function definition* -

$$\begin{tikzcd} \mathbf{pointers} \arrow[r, "\text{square}"] \arrow[d, "*"] & \mathbf{Types} \\ \mathbf{Types} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \text{void} \\ \text{double} & \end{tikzcd} \qquad \begin{tikzcd} y \arrow[r, mapsto, "\text{square}"] \arrow[d, mapsto, "*"] & \text{void} \\ *y & \end{tikzcd}$$

$\Longrightarrow$

```
void square(double *y) { ... }
```

Inside the function definition,

$$\begin{tikzcd} \mathbf{pointer} \arrow[r, "*"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types} \end{tikzcd} \qquad \begin{tikzcd} y \arrow[r, mapsto, "*"] & *y \arrow[r, mapsto, "\text{typedef}"] & \text{double} \end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
*y = x*x
```

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd} \text{function procedure} \arrow[r, bend right] & \mathbf{Types} \arrow[r, "\&"] & \mathbf{Memory} \arrow[d, "\cong"] & \\ & & \mathbf{pointers} \arrow[r, "\text{square}"] & \mathbf{Types} \end{tikzcd}$$

$$\begin{tikzcd} \text{function procedure} \arrow[r, bend right] & \text{double} \arrow[r, "\&"] & \text{Memory}(\text{Obj}\mathit{Memory}) \arrow[d, "\cong"] & \\ & & \text{pointer} \arrow[r, "\text{square}"] & \text{void} \end{tikzcd}$$

$$\begin{tikzcd} \text{function procedure} \arrow[r, bend right] & \text{res} \arrow[r, mapsto, "\&"] & \&\text{res} \arrow[d, mapsto, "\cong"] & \\ & & \&\text{res} \arrow[r, mapsto, "\text{square}"] & \text{square}(\&\text{res}) \end{tikzcd}$$

6.0.2. *C++ syntax for dealing with passing pointers (and arrays) into functions.* However, in *C++*, a lot of the dereferencing $*$ and referencing & is not explicitly said so in the syntax. In this syntax, passing by reference is indicated by prepending the & ampersand to the variable name, in function declaration (prototype and definition). We don't have to explicitly deference the argument in the function (it's done behind the scene) and syntax-wise (it seems), we only have to refer to the argument by regular local name.

Indeed, the syntax appears "shortcutted" greatly:

- *function prototype* -

$$\begin{tikzcd} \mathbf{pointer} \times \mathbf{Types} \arrow[r, "\text{function}"] & \mathbf{Types} \end{tikzcd} \qquad \begin{tikzcd} \text{pointer}, \text{double} \arrow[r, mapsto, "\text{function}"] & \text{void} \end{tikzcd}$$

$\Longrightarrow$

```
void function(double &)
```

- *function definition* -

$$\begin{tikzcd} \mathbf{pointers} \times \mathbf{Types} \arrow[r, "\text{square}"] & \mathbf{Types} \end{tikzcd} \qquad \begin{tikzcd} \text{pointer}, \text{double} \arrow[r, "\text{function}"] & \text{void} \end{tikzcd} \qquad \begin{tikzcd} \&, y \arrow[r, mapsto, "\text{function}"] & \text{function}(\text{double } \&y) \end{tikzcd}$$

$\Longrightarrow$

```
void function(double &y) { ... }
```

Inside the function definition,

$$\begin{tikzcd} \text{double} \arrow[r, "\text{End}(\text{double}, \text{double})"] & \text{double} \end{tikzcd} \qquad \begin{tikzcd} y \arrow[r, mapsto, "\text{End}(\text{double}, \text{double})"] & y = x * x \end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
y = x*x
```

with no deferencing needed.

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd} \mathbf{Types} \arrow[r, "\text{function}"] & \mathbf{Types} \end{tikzcd} \qquad \begin{tikzcd} \text{double} \arrow[r, "\text{function}"] & \text{void} \end{tikzcd} \qquad \begin{tikzcd} \text{res} \arrow[r, mapsto, "\text{function}"] & \text{function}(\text{res}) \end{tikzcd}$$

6.0.3. *C++ note on arrays.* For dealing with arrays, Stroustrup (2013) [7], on pp. 12 of Chapter 1 The Basics, Section 1.8 Pointers, Arrays, and References, does the following:

- *array declaration* -

```
type a[n]; // type[n]; array of n type's
```

- "Using" arrays in function prototypes, i.e. passing into arguments of functions for *function prototypes*

```
data-type function( type * arrayname)
```

- "Using" arrays when "using" functions, i.e. passing into arguments when a function is "called" or "executed"

```
function( arrayname )
```

Fitzpatrick [5] mentions using `inline` for short functions, no more than 3 lines long, because of memory cost of calling a function.

6.0.4. *Need a CUDA, C, C++, IDE? Try Eclipse!* This website has a clear, lucid, and pedagogical tutorial for using Eclipse: Creating Your First C++ Program in Eclipse. But it looks like I had to pay. Other than the well-written tips on the webpage, I looked up stackexchange for my Eclipse questions (I had difficulty with the Eclipse documentation).

Others, like myself, had questions on how to use an IDE like Eclipse when learning CUDA, and "building" (is that the same as compiling?) and running only single files.

My workflow: I have a separate, in my file directory, folder with my github repository clone that's local.

I start a New Project, CUDA Project, in Eclipse. I type up my single file (I right click on the `src` folder and add a 'Source File'). I build it (with the Hammer, Hammer looking icon; yes there are a lot of new icons near the top) and it runs. I can then run it again with the Play, triangle, icon.

I found that if I have more than 1 (2 or more) file in the `src` folder, that requires the `main` function, it won't build right.

So once a file builds and it's good, I, in Terminal, `cp` the file into my local github repository. Note that from there, I could use the `nvcc` compiler to build, from there, if I wanted to.

Now with my file saved (for example, `helloworldkernel.cu`), then I can delete it, without fear, from my, say, `cuda-workplace`, from the right side, "C/C++ Projects" window in Eclipse.

7. On CUDA By Example

Take a look at 3.2.2 A Kernel Call, a Hello World in CUDA C, with a simple kernel, on pp. 23 of Sanders and Kandrot (2010) [8] and on github, [helloworldkernel.cu](https://github.com/ernestyalumni/CompPhys/blob/master/CUDA-By-Example/helloworldkernel.cu). Let's work out the functor interpretation for practice.

- *function definition* -

$$\mathbf{Types} \xrightarrow{\text{kernel}} \mathbf{Types}$$
$$\text{void} \xrightarrow{\text{kernel}} \text{void}$$

  where `kernel` $\in$ `__global__`
  $\Longrightarrow$

```
__global__ void kernel(void) { }
```

  CUDA C adds the `__global__` qualifier to standard C to *alert the compiler that the function*, `kernelfunction`, should be compiled to run on the *device*, not the host (pp. 24 [8]).
- "Using", "calling", "running" function -

  $<<<>>>: (n_{\text{block}}, n_{\text{threads}}) \times \text{kernelfunction} \mapsto \text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>> \in \text{End}(\text{Dat}_{\mathbf{Types}})$
  $<<<>>>: \mathbb{N}^+ \times \mathbb{N}^+ \times \text{Mor}_{\text{GPU}} \to \text{End}(\text{Dat}_{GPU})$

  $\Longrightarrow$

```
kernel<<<1,1>>>();
```

cf. 3.2.3 Passing Parameters of Sanders and Kandrot (2010) [8]

Taking a look at [add-passb.cu](https://github.com/ernestyalumni/CompPhys/blob/master/CUDA-By-Example/add-passb.cu), let's work out the functor interpretation of `cudaMalloc`, `cudaMemcpy`.

In `main`, "declaring" a pointer:

```
int *dev_c
```

$\Longleftarrow$

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "*"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types} \\
\texttt{dev\_c} \arrow[r, mapsto, "*"] & *\texttt{dev\_c} \arrow[r, mapsto, "\text{typedef}"] & \text{int}
\end{tikzcd}$$

We can also do, note, the `sizeof` function (which is a well-defined mapping, for once) on Obj**Types**:

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "*"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types} \arrow[r, "\text{sizeof}"] & \mathbb{N}^+ \\
\texttt{dev\_c} \arrow[r, mapsto, "*"] & *\texttt{dev\_c} \arrow[r, mapsto, "\text{typedef}"] & \text{int} \arrow[r, mapsto, "\text{sizeof}"] & \text{sizeof(int)}
\end{tikzcd}$$

Consider what Sanders and Kandrot says about the pointer to the pointer that (you want to) holds the address of the newly allocated memory. [8] Consider this diagram:

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "*"] & \mathbf{pointers} \arrow[r, "*"] & \mathbf{Types} \\
\mathbf{pointer} \arrow[r, "*"] & \mathbf{pointer} \arrow[r, "*"] & \text{void} \\
\&\texttt{dev\_c} \arrow[r, "*"] & *(\&\texttt{dev\_c}) \arrow[r, "*"] & (\text{void}**)(\&\texttt{dev\_c})
\end{tikzcd}$$

I propose that what `cudaMalloc` does (actually) is the following:

$$\begin{tikzcd}
\mathbf{Memory}_{\text{GPU}} \arrow[r, "\text{cudaMalloc}"] & \mathbf{pointers} \arrow[r, "*"] \arrow[d, "*"] & \mathbf{pointers} \arrow[r, "*"] & \mathbf{Types} \\
 & \mathbf{pointers}_{\text{GPU}} \arrow[r, "*"] & \mathbf{Types} &
\end{tikzcd}$$

(3)

$$\begin{tikzcd}
\text{Memory address}_{\text{GPU}} \arrow[r, mapsto, "\text{cudaMalloc}"] & \&\texttt{dev\_c} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "*"] & *(\&\texttt{dev\_c}) \arrow[r, mapsto, "*"] & (\text{void}**)(\&\texttt{dev\_c}) \\
 & \texttt{dev\_c} \arrow[r, mapsto, "*"] & *\texttt{dev\_c} &
\end{tikzcd}$$

`dev_c` is now a *device pointer*, available to kernel functions on the GPU.

Syntax-wise, we can relate this diagram to the corresponding function "usage":

$$\mathbf{pointers} \times \mathbb{N}^+ \xrightarrow{\text{cudaMalloc}} \texttt{cudaError\_r}$$

$$((\text{void}**)(\&\texttt{dev\_c}), (\text{sizeof(int)})) \overset{\text{cudaMalloc}}{\longmapsto} \text{cudaSuccess (for example)}$$

$\Longrightarrow$

```
cudaMalloc((void**)&dev_c, sizeof(int))
```

For practice, consider now `cudaMemcpy` in the functor interpretation, and its definition as such:

`cudaMemcpy` is a "functor category", s.t. we equip the functor `cudaMemcpy` with a collection of objects $\text{Obj}_{\text{cudaMemcpy}}$, s.t., for example, `cudaMemcpyDevicetoHost` $\in \text{Obj}_{\text{cudaMemcpy}}$, where

$$(\,\text{cudaMemcpy}(-,-,n_{\text{thread}}, \text{cudaMemcpyDevicetoHost}) : \mathbf{Memory}_{GPU} \to \mathbf{Memory}_{CPU}\,) \in \text{Hom}(\mathbf{Memory}_{GPU}, \mathbf{Memory}_{CPU})$$

where $\text{Obj}\mathbf{Memory}_{GPU} \equiv$ collection of all possible memory (addresses) on GPU.

It should be noted that, syntax-wise, $\&c \in \text{Obj}\mathbf{Memory}_{CPU}$ and $\&c$ belongs in the "first slot" of the arguments for cudaMemcpy, whereas `dev_c` $\in \mathbf{pointers}_{GPU}$ a *device pointer*, is "passed in" to the "second slot" of the arguments for cudaMemcpy.

7.1. **Threads, Blocks, Grids.** cf. Chapter 5 Thread Cooperation, Section 5.2. Splitting Parallel Blocks of Sanders and Kandrot (2010) [8].

Consider first a 1-dimensional block.

- `threadIdx.x` $\Longleftarrow M_x \equiv$ number of threads per block in x-direction. Let $j_x = 0 \dots M_x - 1$ be the index for the thread. Note that $1 \leq M_x \leq M_x^{\max}$, e.g. $M_x^{\max} = 1024$, max. threads per block

- `blockIdx.x` $\Longleftarrow N_x \equiv$ number of blocks in x-direction. Let $i_x = 0 \dots N_x - 1$
- `blockDim` stores number of threads along each dimension of the block M_x.

Then if we were to "linearize" or "flatten" in this x-direction,

$$k = j_x + i_x M_x$$

where k is the kth thread. $k = 0 \dots N_x M_x - 1$.

Suppose vector is of length N. So we *need* N parallel threads to launch, in total.
e.g. if $M_x = 128$ threads per block, $N/128 = N/M_x$ blocks to get our total of N threads running.

Wrinkle: integer division! e.g. if $N = 127$, $\frac{N}{128} = 0$.

Solution: consider $\frac{N+127}{128}$ blocks. If $N = l \cdot 128 + r$, $l \in \mathbb{N}$, $r = 0 \dots 127$.

$$\begin{aligned} \frac{N+127}{128} = \frac{l \cdot 128 + r + 127}{128} = \frac{(l+1)128 + r - 1}{128} = \\ = l + 1 + \frac{r-1}{128} = \begin{cases} l & \text{if } r = 0 \\ l+1 & \text{if } r = 1 \dots 127 \end{cases} \end{aligned}$$

$$\begin{aligned} \frac{N+(M_x-1)}{M_x} = \frac{l \cdot M_x + r + M_x - 1}{M_x} = \frac{(l+1)M_x + r - 1}{M_x} = \\ = l + 1 + \frac{r-1}{M_x} = \begin{cases} l & \text{if } r = 0 \\ l+1 & \text{if } r = 1 \dots M_x - 1 \end{cases} \end{aligned}$$

So $\frac{N+(M_x-1)}{M_x}$ is the smallest multiple of M_x greater than or equal to N, so $\frac{N+(M_x-1)}{M_x}$ **blocks are needed or more than needed to run a total of N threads.**

Problem: Max. grid dim. in 1-direction is 65535, $\equiv N_i^{\max}$.
So $\frac{N+(M_x-1)}{M_x} = N_i^{\max} \Longrightarrow N = N_i^{\max} M_x - (M_x - 1) \leq N_i^{\max} M_x$. i.e. number of threads N is limited by $N_i^{\max} M_x$.
Solution.

- number of threads per block in x-direction $\equiv M_x \Longrightarrow$ `blockDim.x`

- number of blocks in grid $\equiv N_x \Longrightarrow$ `gridDim.x`
- $N_x M_x$ total number of threads in x-direction. Increment by $N_x M_x$. So next scheduled execution by GPU at the $k = N_x M_x$ thread.

7.2. (CUDA) Constant Memory.

cf. Chapter 6 Constant Memory of Sanders and Kandrot (2010) [8]

Refer to the ray tracing examples in Sanders and Kandrot (2010) [8], and specifically, here: raytrace.cu, rayconst.cu.

Without constant memory, then this had to be done:

- *definition* (in the code) - Consider **struct** as a subcategory of **Types** since **struct** itself is a category, equipped with objects and functions (i.e. methods, modules, etc.).

  So for **struct**, Obj**struct** $\ni$ `Sphere`. $\Longrightarrow$

  ```
  struct sphere { ... }
  ```

- Usage, "instantiation", i.e. creating, or "making" it (the `struct`):

$$\begin{tikzcd} \textbf{pointers} \arrow[r, "*"] & \textbf{Types} \end{tikzcd}$$

$$\begin{tikzcd} \textbf{pointer} \arrow[r, "*"] \arrow[d, "\&"'] & \textbf{struct} \\ \textbf{Memory}_{CPU} & \end{tikzcd}$$

$$\begin{tikzcd} s \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"'] & *s \arrow[r, mapsto, "\text{typedef}"] & \texttt{Sphere} \\ \textbf{Memory}\text{address}_{CPU} & & \end{tikzcd}$$

$\Longrightarrow$

```
Sphere *s
```

Recalling Eq. 3, for `SPHERES` == 40 (i.e. for example, 40 spheres)

```
cudaMalloc((void **) &s, sizeof(Sphere)*SPHERES)
```

$\Longleftarrow$

$$\begin{tikzcd} \textbf{Memory}_{\text{GPU}} \arrow[r, "\text{cudaMalloc}"] & \textbf{pointers} \arrow[r, "*"] \arrow[d, "*"] & \textbf{pointers} \arrow[r, "*"] & \textbf{Types} \\ & \textbf{pointers}_{\text{GPU}} \arrow[r, "*"] & \textbf{Types} & \end{tikzcd}$$

$$\begin{tikzcd} \text{Memory address}_{\text{GPU}} \arrow[r, mapsto, "\text{cudaMalloc}"] & \&s \arrow[r, mapsto, "*"] \arrow[d, mapsto, "*"] & *(\&s) \arrow[r, mapsto, "*"] & (\text{void} **)(\&s) \\ & s \arrow[r, mapsto, "*"] & *s & \end{tikzcd}$$

and syntax-wise,

$$\textbf{pointers} \times \mathbb{N}^+ \xrightarrow{\text{cudaMalloc}} \texttt{cudaError\_r}$$

$$((\text{void} **)(s), \text{sizeof}(\text{Sphere}) * \text{SPHERES}) \xmapsto{\text{cudaMalloc}} \text{cudaSuccess (for example)}$$

Now consider

```
cudaMemcpy(s, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice)
```

$$\begin{tikzcd}
\textbf{Memory}_{CPU} \arrow[r, "\text{cudaMemcpy}(s, temps, \text{sizeof(Sphere)} * \text{SPHERES}, \text{cudaMemcpyHostToDevice})"] & \textbf{Memory}_{GPU} \\
\textbf{Memory}\text{address}_{CPU} \arrow[r, mapsto] \arrow[d, mapsto, "*"'] & \textbf{Memory}\text{address}_{GPU} \arrow[d, mapsto, "*"'] \\
\texttt{temp\_s} \arrow[r, mapsto] & s
\end{tikzcd}$$

The lesson then is this, in light of how long ray tracing takes with constant memory and without constant memory `-` `cudaMemcpy` between host to device, CPU to GPU, is a costly operation. Here, in this case, we're copying from the host memory to memory on the GPU. It copies to a global memory on the GPU.

Now, using **constant memory**,
we no longer need to do `cudaMalloc`, allocate memory on the GPU, for s, pointer to a `Sphere`.

Instead, we have

```
__constant__ Sphere s[SPHERES];
```

In this particular case, we want it to have global scope.
Note, it is still on host memory.

Notice that

$$\begin{tikzcd}
\textbf{Memory}_{CPU} \arrow[r, "\text{cudaMemcpyHostToDevice}(-, -, \text{sizeof(Sphere)} * \text{SPHERES})"] & \textbf{Memory}_{GPU}
\end{tikzcd}$$

$$\begin{tikzcd}
\textbf{Memory}\text{address}_{CPU} \arrow[r, mapsto, "\text{cudaMemcpyHostToDevice}(-, -, \text{sizeof(Sphere)} * \text{SPHERES})"] \arrow[d, mapsto, "*"'] & \textbf{Memory}\text{adddress}_{GPU} \arrow[d, mapsto, "*"'] \\
\texttt{temp\_s} \arrow[r, mapsto] \arrow[d, mapsto, "\text{typedef}"'] & s \arrow[d, mapsto, "\text{typedef}"'] \\
\text{array of Sphere's} \arrow[r, mapsto] & \text{array of Sphere's}
\end{tikzcd}$$

So notice that we have a bijection, and on one level, we can think of the bijection from `temp_s`, an array of Sphere's to s, an array of Sphere's. So notice that the types and memory size of `temp_s` and s must match.

And for this case, that's all there is to *constant memory*. What's going on involves the so-called *warp*, a collection of threads, "woven together" and get executed in lockstep. NVIDIA hardware broadcasts a single memory read to each half-warp. "If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread." (cf. Sanders and Kandrot (2010) [8]). Furthermore, "the hardware can aggressively cache the constant data on the GPU."

References

[1] Trevor Hastie, Robert Tibshirani, Jerome Friedman. **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**, Second Edition (Springer Series in Statistics) 2nd ed. 2009. Corr. 7th printing 2013 Edition. ISBN-13: 978-0387848570. https://web.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf

[2] Jared Culbertson, Kirk Sturtz. *Bayesian machine learning via category theory.* arXiv:1312.1445 [math.CT]

[3] John Owens. David Luebki. *Intro to Parallel Programming. CS344.* **Udacity** http://arxiv.org/abs/1312.1445 Also, https://github.com/udacity/cs344

[4] CS229 Stanford University. http://cs229.stanford.edu/materials.html

[5] Richard Fitzpatrick. "Computational Physics." http://farside.ph.utexas.edu/teaching/329/329.pdf

[6] M. Hjorth-Jensen, **Computational Physics**, University of Oslo (2015) http://www.mn.uio.no/fysikk/english/people/aca/mhjensen/

[7] Bjarne Stroustrup. **A Tour of C++** (C++ In-Depth Series). Addison-Wesley Professional. 2013.

[8] Jason Sanders, Edward Kandrot. **CUDA by Example: An Introduction to General-Purpose GPU Programming** 1st Edition. Addison-Wesley Professional; 1 edition (July 29, 2010). ISBN-13: 978-0131387683