TEXTURE MEMORY IN CUDA | WHAT IS TEXTURE MEMORY IN CUDA PROGRAMMING

POSTED BY NITIN GUPTA AT 22:02 | 4 COMMENTS

Texture Memory in CUDA | What is Texture Memory in CUDA programming

We have talked about the global memory, shared memory and constant Memory in previous article(s), we also some example like Vector Dot Product, which demonstrate how to use shared memory. CUDA architecture provides another kind of memory which we call Texture Memory. In this article, we learn about texture memory and we answer following questions

1. What is Texture memory in CUDA?
2. Why Texture memory?
3. Where the Texture memory resides?
4. How does Texture memory work's in CUDA?
5. How to use Texture memory in CUDA?
6. Where to use and where should not use Texture memory in CUDA?
7. Example of using texture memory in CUDA, step by step.
8. Performance consideration of Texture memory.

I'll present an example at the end of the article, so keep reading because for understanding texture memory basics is very important.

Motivation

When we looked at constant Memory, we saw how exploiting special memory spaces under the right circumstances can dramatically accelerate applications. We also learned how to measure these performance gains in order to make informed decisions about performance choices. In this article, we will learn about how to allocate and use ***texture memory***. Like constant memory, texture memory is another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Although texture memory was originally designed for traditional graphics applications, it can also be used quite effectively in some GPU computing applications.

TAGS

CUDA ADVANCE

CUDA PROGRAMMING CONCEPT

OPTIMIZATION IN CUDA

RELATED POSTS

Using Shared Memory in CUDA C/C++

Optimization in histogram CUDA code: When number of Bins not equal to max threads in a block

Further Optimization in histogram CUDA code | Fast implementation of histogram in CUDA

Optimizing histogram CUDA code | Optimization in histogram in CUDA

Computing Histogram on CUDA | CUDA code for Histogram computation | Computing Histogram on GPU using CUDA

SHARE THIS

RECENT POPULAR RANDOM

TEXTURE OBJECT IN CUDA | Bindless Texture in CUDA
01/04/2013 - 10 comments

How to specify architecture while compiling CUDA program in Visual profiler
How to change command line flag in CUDA
24/03/2013 - 0 comments

Using Shared Memory in CUDA C/C++
11/03/2013 - 2 comments

Optimization in histogram CUDA code: When number of Bins not equal to max threads in a block
08/03/2013 - 9 comments

GOOGLE+ FOLLOWERS

Nitin Gupta

Add to circles

67 have me in circles

View all

Article Objective

In the course of this article, you will accomplish the following;

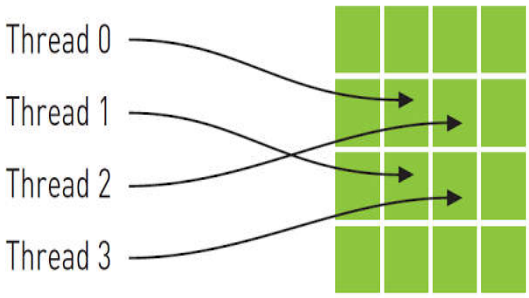
- You will learn about the performance characteristics of texture memory.
- You will learn how to use one-dimensional texture memory with CUDA C.
- You will learn how to use Two-dimensional texture memory with CUDA C.

Don't worry; we also answer all the above questions that we have previously mentioned.

Why / (What is) Texture memory in CUDA?

If you read the motivation to this article, the secret is already out: There is yet another type of read-only memory that is available for use in your programs written in CUDA C. Readers familiar with the workings of graphics hardware will not be surprised, but the GPU's sophisticated *texture memory* may also be used for general-purpose computing. Although NVIDIA designed the texture units for the classical OpenGL and DirectX rendering pipelines, texture memory has some properties that make it extremely useful for computing.

Like constant memory, **texture memory is cached on chip**, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of ***spatial locality***. **In a computing application, this roughly implies that a thread is likely to read from an address "near" the address that nearby threads read, as shown in Figure**



Note:
Arithmetically, the four addresses shown are not consecutive, so they would not be cached together in a typical CPU caching scheme. But since GPU texture caches are designed to accelerate access patterns such as this one, you will see an increase in performance in this case when using texture memory instead of global memory.

Let's go in Deep

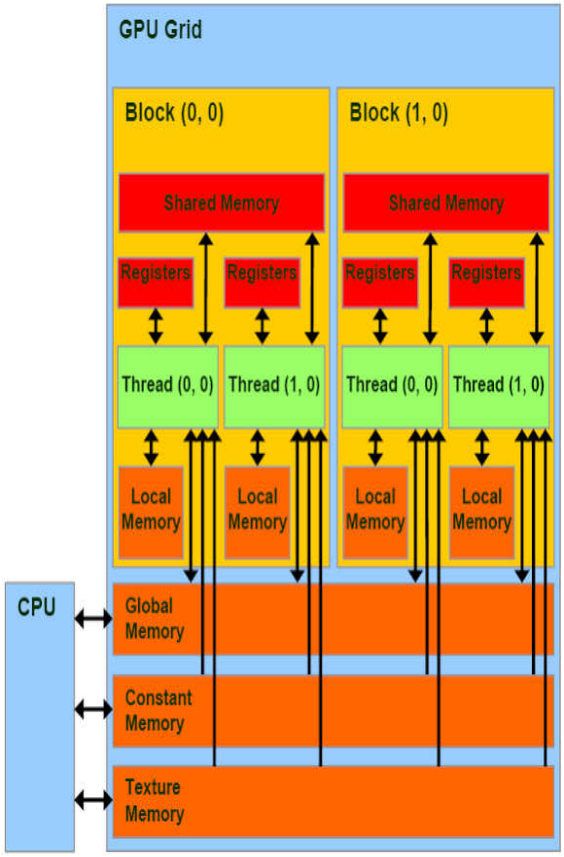
The read-only texture memory space is cached. Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache. **The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.** Texture memory is also designed for streaming fetches with a constant latency; that is, a cache hit reduces DRAM bandwidth demand, but not fetch latency.

In certain addressing situations, reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant memory.

Note:
For Spatial Locality
Link 1 [Overview of spatial locality]
Link 2 [Detailed description on spatial locality with example]

Where the Texture memory resides

Following fig. will let you know where the texture memory resides in CUDA architecture.



How does Texture memory work's and how to use texture memory in CUDA

Texture memory is read from kernels using the device functions described at here.

Reading the Texture memory

The process of reading a texture is called a ***texture fetch***. The first parameter of a texture fetch specifies an object called a ***texture reference***.
A texture reference defines which part of texture memory is fetched. As detailed in Section 3.2.10.1.3 in CUDA programming guide 4.2; it must be bound through runtime functions to some region of memory, called a ***texture***, before it can be used by a kernel. Several distinct texture references might be bound to the same texture or to textures that overlap in memory.

A texture reference has several attributes. One of them is its **dimensionality** that specifies whether the texture is addressed as a one-dimensional array using one ***texture coordinate***, a two-dimensional array using two texture coordinates, or a three-dimensional array using three texture coordinates. **Elements of the array are called *texels***, short for "texture elements." The type of a Texel is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types defined in Section B.3.1 in CUDA programming guide 4.2).
Other attributes define the input and output data types of the texture fetch, as well as how the input coordinates are interpreted and what processing should be done.
A texture can be any region of linear memory or a CUDA array (described in Section 3.2.10.2.3 in CUDA programming guide 4.2). Table F-2 lists the maximum texture width, height, and depth depending on the compute capability of the device.
Textures can also be layered as described in Section 3.2.10.1.5. CUDA programming guide 4.2

cudaChannelFormatDesc () in CUDA | How to use cuda...

Texture Memory in CUDA | What is Texture Memory in...

► January (41)

► 2012 (15)

A point to remember regarding Texture memory

Within a kernel call, the texture cache is not kept coherent with respect to global memory writes, so texture fetches from addresses that have been written via global stores in the same kernel call return undefined data. That is, a thread can safely read a memory location via texture if the location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread within the same kernel call. This is relevant only when fetching from linear or pitch-linear memory because **a kernel cannot write to CUDA arrays**.

Note:-
If textures are fetched using **tex1D(), tex2D(), or tex3D()** rather than **tex1Dfetch(),** the hardware provides other capabilities that might be useful for some applications such as image processing,

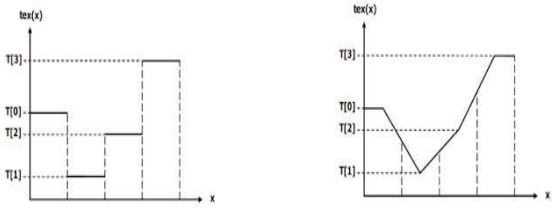
More details can be found at here

Nice Texture Memory Features / Advantages of Texture memory in CUDA

I would like to discuss some basic features of texture memory before start looking how to use in your CUDA code. From NVidia, they provide some very awesome feature alongside it inherits some nice features from the graphics pipeline.
Feature of texture memory for programmer convince, here some of them,

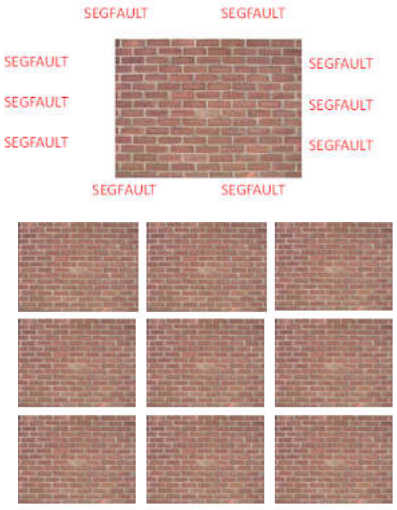
Get some things for free:

- Linear interpolation of adjacent data values



- Automatic normalization of data when you fetch it
  - [0,1] for unsigned values
  - [-1,1] for signed values
- Automatic normalization of array indices to [0,1]
  - e.g. same code for all problem sizes
- Automatic boundary handling

Fig 3



Disadvantages of Texture memory in CUDA

- Read-only
  - Can only update 3D textures by performing a memcpy to some rectangular region of the texture
  - However if your texture is 2D, you can write directly into it via a *surface* object (go through CUDA programming guide 4.2)

When to Use Texture Memory in CUDA

- If you update your data rarely but read it often...
- ...especially if there tends to be some kind of spatial locality to the read access pattern...
  - i.e. nearby threads access nearby locations in the texture
- ...and especially if the precise read access pattern is difficult to predict
- Also, you need to use texture memory in order to visualize your data using the graphics pipeline (consult the **CUDA SDK** samples)

When Not to Use Texture Memory in CUDA

We should not use texture memory when we read our input data exactly once after update it.

Example of texture memory in CUDA

For understanding example please go through the following function, which we are going to use in this example to demonstrate how to use texture memory in CUDA, functions are;

cudaBindTexture, cudaBindTexture2D, cudaBindTexture2D, cudaUnbindTexture cudaBindTextureToArray, cudaCreateChannelDesc **and** Texture functions (Texture object and Reference API)

I have already explained that, the process of reading a texture is called a ***texture fetch***. The first parameter of a texture fetch specifies an object called a ***texture reference***. A texture reference defines which part of texture memory is fetched.

Steps of using Texture in Your CUDA code

Now we learn how to use texture memory in your CUDA C code. I'll illustrate this step by step.

Overview of steps

- *Declare the texture memory in CUDA.*
- *Bind the texture memory to your texture reference in CUDA.*
- *Read the texture memory from your texture reference in CUDA Kernel.*
- *Unbind the the texture memory from your texture reference in CUDA.*

Step 1

Declaration of texture memory in CUDA

```
Texture (<type>, <dim>, <readmode>)
<texture_reference>
```

• **<texture_reference>** is a *handle* tied to **compile-time** attributes of some texture array. Declaration DOES NOT

- Allocate storage
- Associate storage with the handle
-

• **<type>** is type of **texel** data returned from an access to the texture

- Basic integer or floating types
- 1, 2, or 4 element builtin vector types

• **<dim>** is dimensionality of texture array

- 1, 2, or 3; optional – if not present then assumed 1

• **<readmode>** controls conversion of **texel** returned by an access

- ***cudaReadModeElementType***: no conversion performed
- ***cudeReadModeNormalizedFloat*** if type is integer, value returned is mapped to [-1.0,1.0] for signed, and [0.0, 1.0] for unsigned

Note:

Optional – defaults to **cudaReadModeElementType**

Example

```
texture <float> texture_reference ;
```

We have declared a texture of name "texture_reference" of type "float" of dimension "1D" and readmode is "**cudaReadModeElementType**"

Step 2

Binding texture memory to your texture reference in CUDA

Now we bind the texture reference to the memory buffer using **cudaBindTexture ()**. This basically tells to CUDA runtime two things;

- We intend to use the specified buffer as a texture.
- We intend to use the specified texture reference as the texture's "name"

Format of cudaBindtexture

```
cudaBindtexture (size *t offset, const struct
texture<T, dim, readMode> & tex , const void *
devptr, size_t size= UINT_MAX) ;
```

Binds `size` bytes of the memory area pointed to by **`devPtr`** to texture reference **`tex`**. The channel descriptor is inherited from the texture reference type. The `offset` parameter is an optional byte offset as with the low-level cudaBindTexture(size_t*, const struct textureReference*, const void*, const struct cudaChannelFormatDesc*, size_t) function. **Any memory previously bound to `tex` is unbound.**

Parameters:

offset - Offset in bytes

tex - Texture to bind

devPtr - Memory area on device

size - Size of the memory area pointed to by devPtr

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Example

```
HANDLE_ERROR ( cudaBindTexture( NULL,
texture_reference, tobereference , sizeinbytes ) );
```

Here we bind "tobereference" memory to "texture_reference" starting from initially, specified by "NULL"(offset), and "sizeinbytes" in size of memory, which is the size of memory referenced by "tobereference".

At this point, our textures are completely set up, and we're ready to launch our kernel.

Step 3

Reading texture memory from your texture reference in CUDA Kernel

When we're reading from textures in the kernel, we need to use special functions to instruct the GPU to route our requests through the texture unit and not

through standard global memory. As a result, we can no longer simply use square brackets to read from buffers; we need to modify kernel function to use tex1Dfetch () when reading from memory.

Additionally, there is another thing to be note in texture memory reading; since in all the previous posts under CUDA programming section we were using global memory to read data as either input or output. Now we need to change this scenario and use texture memory that requires us to make some other changes too. Although it looks like a function, tex1Dfetch () **is a compiler intrinsic. And since texture references must be declared globally at file scope, we can no longer pass the input and output buffers as parameters to our kernel function because the compiler needs to know at compile time which textures** tex1Dfetch () **is should be sampling.**

Rather than passing pointers to input and output buffers as we previously did in previous post's in CUDA programming section, we will pass to kernel function a **Boolean flag "indicator" that indicates which buffer to use as input and which to use as output.**

Format of tex1Dfetch ()

Older version [format has taken from this **link]**
tex1Dfetch - performs an unfiltered texture lookup in a given sampler.

Synopsis

```
float4 tex1Dfetch (sampler1D samp, int4 S) ;
float4 tex1Dfetch (sampler1D samp, int4 S , int texeloff) ;
int4 tex1Dfetch (isampler1D samp, int4 S) ;
int4 tex1Dfetch (isampler1D samp, int4 S , int texeloff) ;
unsigned int4 tex1Dfetch (isampler1D samp, int4 S) ;
unsigned int4 tex1Dfetch (isampler1D samp, int4 S , int texeloff) ;
```

Parameters

Samp: Sampler to lookup.
S: Coordinates to perform the lookup. The level of detail is stored in the last component of the coordinate vector.
texelOff: Offset to be added to obtain the final texel.

Description

Performs an unfiltered texture lookup in sampler ***samp*** using coordinate **S**. The level of detail is provided by the last component of the coordinate vector. May **use texel offset *texelOff*** to compute final texel.

Profile Support

tex1Dfetch is only supported in gp4 and newer profiles.

Newer Version [Format has been taken from CUDA programming guide 4.2 **]**

```
Template <class DataType>
Type tex1Dfetch( texture <DataType, cudaTextureType1D,cudaReadModeElementType> texRef, int x )
;
```

Example with different DataType

```
float tex1Dfetch( texture<unsigned char, cudaTextureType1D,
cudaReadModeNormalizedFloat> texRef,int x);
```

```
float tex1Dfetch(texture<signed char, cudaTextureType1D,
cudaReadModeNormalizedFloat> texRef, int x);
```

```
float tex1Dfetch(texture<unsigned short, cudaTextureType1D,
cudaReadModeNormalizedFloat> texRef,int x);
```

```
float tex1Dfetch( texture<signed short, cudaTextureType1D,
cudaReadModeNormalizedFloat> texRef, int x);
```

Important

It fetch the region of linear memory bound to texture reference **texRef** using integer texture coordinate **x**. **tex1Dfetch()** only works with non-normalized coordinates (Section 3.2.10.1.2), so only the border and clamp addressing modes are supported. **It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point.**

Example of use

t = tex1Dfetch (texture_reference coordinate_x);
It reads the pixel from image reference by "texture_reference" at the coordinate "x" and value get assigned to "t".

Step 3 : Final step at the end

At the end of the application Rather than just freeing the global buffers, we also need to **unbind textures:**

Unbinding the texture memory from your texture reference in CUDA

*cudaError_t cudaUnbindTexture (const struct textureReference *texRef) ;*

This will unbind your texture from texture memory.

Example

HANDLE_ERROR (*cudaUnbindTexture (texture_reference) ;*
This will unbind your texture_reference from texture memory.

Performance consideration of Texture memory

Texture memory can provide additional speedups if we utilize some of the conversions that texture samplers can perform automatically, such as unpacking packed data into separate variables or **converting 8- and 16-bit integers to normalized floating-point numbers**. We didn't explore either of these capabilities in this article, but they might be useful to you! Stay tuned!!

Somewhat important

If you have understood texture memory perfectly then you can skip this.
Be hands on this documentation, this will describe everything of Texture memory.

If you are looking at complete example, you must like this example, Stay tuned!!

Summary of this article

As we saw in the previous article with constant Memory, some of the benefit of texture memory comes as the result of on-chip caching. This is especially noticeable in applications like image processing: applications that have some spatial coherence to their data access patterns.
We talked about

- What is Texture memory in CUDA?
- Why Texture memory?
- Where the Texture memory resides?
- How does Texture memory work's in CUDA?
- How to use Texture memory in CUDA?
- Where to use and where should not use Texture memory in CUDA?
- Example of using texture memory in CUDA, step by step.
- Performance consideration of Texture memory.

I hope you must like this article and have learned texture memory and how to use texture memory in CUDA.

Got Questions?
Feel free to ask me any question because I'd be happy to walk you through step by step!
Want to Contact us? Click here

4 comments:

jyoti soni 20 May 2014 at 23:11

Great post . It takes me almost half an hour to read the whole post. Definitely this one of the informative and useful post to me. Thanks for the share and plz visit my site Programming 21st and century provide offshore outsourcing service, freelance service, personal assistant,web development, customer service, marketing services.

Reply

Anonymous 3 December 2014 at 06:17

thank you for these details, it's very interesting very useful, I have a question I am trying to program an algorithm in CUDA C, I work on a data vector that I update after each iteration, what type memory can I use to reduce the execution time ? thank you

Reply

Anonymous 16 April 2015 at 13:13

Thank you for the clear articles in simple language

Reply

melbourne web developer 29 October 2015 at 20:26

thank you for sharing this information, it very useful

Reply

Enter your comment...

Comment as: Unknown (G ▼ Sign out

Publish Preview Notify me

Help us to improve our quality and become contributor to our blog

Links to this post

Create a Link

Newer Post Home Older Post

Become a contributor to this blog. Click on contact us tab

LABELS LIKE US ADMIN

09:02:10 pm

CLOUD

Blogumulus by Roy Tanck and Amanda Fazani

Nitin Gupta

Name:
Nitin Gupta
Email:
nitinguptaiit@gmail.com
Status:
None