CSE 591: GPU Programming

Memories: Constant and Texture

Klaus Mueller

Computer Science Department
Stony Brook University

Code examples from Shane Cook “CUDA Programming”
What is it?

- a *virtual* addressing of global memory
- no specially reserved constant memory space
- read-only

How can you use it?

- declare at compile time by `__constant__` keyword (fastest)
- write at run time using `cudaCopyToSymbol()` before kernel invocation

How can it help you?

- it is cached and so enables single-cycle access when data is in cache
- can broadcast a single value to all the threads within a warp
Constant Memory Caching

Compute 1.x devices (pre-Fermi)

• per SM: 64K block with 8K cache size
• break program into 8K chunks to make best use of cache

Avoid irregular data accesses with bad locality

• not a good use of constant memory
• incurs global memory access overhead + cache write/access overhead

Break program into 64K sized tiles or even better, 8K tiles

• makes efficient use of cache
Constant Memory Caching

Compute 2.x devices

• here any constant section of data can be treated as constant memory
• use keyword const and access will go through the constant cache
• but it has to be non-thread → indexing must not include threadIdx.x

For per-thread access

• need to declare with __constant__ at compile time
• or use cudaCopyToSymbol() before kernel invocation

One thing to keep in mind

• these devices also have L2 cache (much larger than constant cache)
• L2 cache might collect some information automatically
  example: boundary (halo) cells of tiles needed for processing other tiles
• might work quicker in some cases
• also saves memory
Constant Memory Broadcast

Also supported in L2 cache

Comes in handy when all threads require the same data item
  • for example: rotation matrix in graphics
  convolution mask in image processing
  • this item is provided in a single cycle to all the threads in the warp

Be aware of literals
  • will be wasteful to keep in constant memory
  • rather use `#define` statement
  • for example: `#define PI 3.14159265359` (or more digits)
  • or `d += 676136.89`
  • speed is the same, but memory storage is different

For the following program
  • `&`, `|`, and `^` are bitwise AND, OR, and XOR operators
__global__ void const_test_gpu_literal(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = 0x55555555;

        for (int i=0;i<KERNEL_LOOP;i++)
        {
            d ^= 0x55555555;
            d |= 0x77777777;
            d &= 0x33333333;
            d |= 0x11111111;
        }

        data[tid] = d;
    }
}
__global__ void const_test_gpu_const(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = const_data_01;

        for (int i=0;i<KERNEL_LOOP;i++)
        {
            d ^= const_data_01;
            d |= const_data_02;
            d &= const_data_03;
            d |= const_data_04;
        }
        data[tid] = d;
    }
}

#define KERNEL_LOOP 65536

__constant__ static const u32 const_data_01 = 0x55555555;
__constant__ static const u32 const_data_02 = 0x77777777;
__constant__ static const u32 const_data_03 = 0x33333333;
__constant__ static const u32 const_data_04 = 0x11111111;
__global__ void const_test_gpu_gmem(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = 0x55555555;

        for (int i=0; i<KERNEL_LOOP; i++)
        {
            d ^= data_01;
            d |= data_02;
            d &= data_03;
            d |= data_04;
        }

        data[tid] = d;
    }
}

__device__ static u32 data_01 = 0x55555555;
__device__ static u32 data_02 = 0x77777777;
__device__ static u32 data_03 = 0x33333333;
__device__ static u32 data_04 = 0x11111111;
Compare literal with constant memory

- only small differences, neck to neck race

  ID:3 GeForce GTX 460: Literal version is faster by: 0.59ms (C=541.43ms, L=542.02ms)
  ID:3 GeForce GTX 460: Literal version is faster by: 0.17ms (C=541.20ms, L=541.37ms)
  ID:3 GeForce GTX 460: Constant version is faster by: 0.45ms (C=542.29ms, L=541.83ms)
  ID:3 GeForce GTX 460: Constant version is faster by: 0.27ms (C=542.17ms, L=541.89ms)
  ID:3 GeForce GTX 460: Constant version is faster by: 1.17ms (C=543.55ms, L=542.38ms)
  ID:3 GeForce GTX 460: Constant version is faster by: 0.24ms (C=542.92ms, L=542.68ms)
Compare literal with global memory

Speedups:

- compute 1.1 hardware (9800GT): 40:1
- compute 1.3 hardware (GTX260): 3:1
- compute 2.0 hardware (GTX470): 1.8:1
- compute 2.1 hardware (GTX460): 1.6:1
- appears that even L2 devices can benefit from constant memory
Further Examination

It is interesting that constant and literal was equal speed
• shouldn’t constant be slower?

Also, it is interesting that global was slower for all hardware
• shouldn’t devices with L2 cache be better?

Have a look at PTX code
• it is a good habit to look at PTX code when something is unclear
• common practice among experience GPU programmers
PTX Code for Constant Kernel (1)

```plaintext
.const .u32 const_data_01 = 1431655765;
.const .u32 const_data_02 = 2004318071;
.const .u32 const_data_03 = 858993459;
.const .u32 const_data_04 = 286331153;

.entry _Z20const_test_gpu_constPjj (  
    .param .u64 __cudaparm__Z20const_test_gpu_constPjj_data,  
    .param .u32 __cudaparm__Z20const_test_gpu_constPjj_num_elements)
{
    .reg .u32 %r<29>;
    .reg .u64 %rd<6>;
    .reg .pred %p<5>;
    // __cuda_local_var_108907_15_non_const_tid = 0
    // __cuda_local_var_108910_13_non_const_d = 4
    // i = 8
    .loc 16 40 0
```
$LDWbegin__Z20const_test_gnu_constPjj:
$LDWbeginblock_181_1:
  .loc 16 42 0
  mov.u32 %r1, %tid.x;
  mov.u32 %r2, %ctaid.x;
  mov.u32 %r3, %ntid.x;
  mul.lo.u32 %r4, %r2, %r3;
  add.u32 %r5, %r1, %r4;
  mov.s32 %r6, %r5;
  .loc 16 43 0
  ld.param.u32

[__cudaparm__Z20const_test_gnu_constPjj_num_elements];
  mov.s32 %r8, %r6;
  setp.le.u32 %p1, %r7, %r8;
  @%p1 bra $L_1_3074;
$LDWbeginblock_181_3:
  .loc 16 45 0
  mov.u32 %r9, 1431655765;
  mov.s32 %r10, %r9;
$LDWbeginblock_181_5:
  .loc 16 47 0
  mov.s32 %r11, 0;
  mov.s32 %r12, %r11;
  mov.s32 %r13, %r12;
  mov.u32 %r14, 4095;
  setp.gt.s32 %p2, %r13, %r14;
  @%p2 bra $L_1_3586;

$L_1_3330:
  .loc 16 49 0
  mov.s32 %r15, %r10;
  xor.b32 %r16, %r15, 1431655765;
  mov.s32 %r10, %r16;
  .loc 16 50 0
  mov.s32 %r17, %r10;
  or.b32 %r18, %r17, 2004316071;
  mov.s32 %r10, %r18;
  .loc 16 51 0
  mov.s32 %r19, %r10;
  and.b32 %r20, %r19, 858993459;
  mov.s32 %r10, %r20;
  .loc 16 52 0
  mov.s32 %r21, %r10;
  or.b32 %r22, %r21, 206331153;
  mov.s32 %r10, %r22;
  .loc 16 47 0
  mov.s32 %r23, %r12;
  add.s32 %r24, %r23, 1;
  mov.s32 %r12, %r24;
$Lt_1_1794:
  mov.s32 %r25, %r12;
  mov.u32 %r26, 4095;
  setp.le.s32 %p3, %r25, %r26;
  @%p3 bra $L_1_3330;
  @%p3 bra $L_1_3586;
$L_1_3586:
$LDWendblock_181_5:
    .loc 16 55 0
    mov.s32  %r27, %r10;
    ld.param.u64  %rd1, [__cudaparm__Z20const_test_gpu_constPjj_data];
    cvt.u64.u32  %rd2, %r6;
    mul.wide.u32  %rd3, %r6, 4;
    add.u64  %rd4, %rd1, %rd3;
    st.global.u32  [%rd4+0], %r27;
$L_DWendblock_181_3:
$L_1_3074:
$LDWendblock_181_1:
    .loc 16 57 0
    exit;
$LDWend__Z20const_test_gpu_constPjj:
} // __Z20const_test_gpu_constPjj
Discussion of PTX Code

Compiler smartly converted constants into literals
  • so the difference is hidden

Let’s have a look at the global memory version

```
ld.global.u32 %r16, [data_01];
xor.b32 %r17, %r15, %r16;
```

• now an actual memory read occurs
• supported by L2 cache in new architectures

How can we really test the constant memory caching?
  • declare the constant version as an array

```
__constant__ static const u32 const_data[4] = { 0x55555555, 0x77777777, 0x33333333, 0x11111111 };
```
```c
__global__ void const_test_gpu_const(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = const_data[0];

        for (int i=0; i<KERNEL_LOOP; i++)
        {
            d ^= const_data[0];
            d |= const_data[1];
            d &= const_data[2];
            d |= const_data[3];
        }

        data[tid] = d;
    }
}
```
ld.const.u32 %r15, [const_data+0];
mov.s32 %r16, %r10;
xor.b32 %r17, %r15, %r16;
mov.s32 %r10, %r17;
.loc 16 47 0
ld.const.u32 %r18, [const_data+4];
mov.s32 %r19, %r10;
or.b32 %r20, %r18, %r19;
mov.s32 %r10, %r20;
**Discussion of PTX Code**

<table>
<thead>
<tr>
<th>ID</th>
<th>GPU Model</th>
<th>Memory Type</th>
<th>Time Difference</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1496.41ms</td>
<td>1565.96ms, 69.55ms</td>
</tr>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1496.72ms</td>
<td>1566.42ms, 69.71ms</td>
</tr>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1498.14ms</td>
<td>1567.78ms, 69.64ms</td>
</tr>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1496.12ms</td>
<td>1565.81ms, 69.69ms</td>
</tr>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1496.91ms</td>
<td>1566.61ms, 69.70ms</td>
</tr>
<tr>
<td>1</td>
<td>GeForce 9800</td>
<td>Constant</td>
<td>1495.76ms</td>
<td>1565.49ms, 69.73ms</td>
</tr>
<tr>
<td>3</td>
<td>GeForce GTX 460</td>
<td>GMEM</td>
<td>0.20ms</td>
<td>54.18ms, 54.38ms</td>
</tr>
<tr>
<td>3</td>
<td>GeForce GTX 460</td>
<td>GMEM</td>
<td>0.17ms</td>
<td>54.86ms, 55.03ms</td>
</tr>
<tr>
<td>3</td>
<td>GeForce GTX 460</td>
<td>GMEM</td>
<td>0.25ms</td>
<td>54.83ms, 55.07ms</td>
</tr>
<tr>
<td>3</td>
<td>GeForce GTX 460</td>
<td>GMEM</td>
<td>0.81ms</td>
<td>54.24ms, 55.05ms</td>
</tr>
<tr>
<td>3</td>
<td>GeForce GTX 460</td>
<td>Constant</td>
<td>1.14ms</td>
<td>54.83ms, 53.69ms</td>
</tr>
</tbody>
</table>

**Observations**

- Without L2 cache, the winner is constant memory.
- With L2 cache, the winner is global memory.
Lessons Learned

Always check out PTX code to get insight

Be creative to get the insight you want
  • here we changed into an array although we did not need to
  • applied the knowledge that arrays never become literals
  • it can be a detective problem
Recall

- constant memory is not really constant memory
- there is no dedicated special area of memory set aside
- the 64 K limit is exactly a 16-bit offset
- this allows very quick 16-bit addressing

Constant memory can be updated in chunks/tiles at run time

- of up to 64K at a time
- use `cudaCopyToSymbol()` API call
- but this can be expensive – do not do this in loops!
Create some data on the host and use the API call

```c
generate_rand_data(const_data_host);
```

- the constant memory case

```c
// Copy host memory to constant memory section in GPU
CUDA_CALL(cudaMemcpyToSymbol(const_data_gpu, const_data_host,
    KERNEL_LOOP * sizeof(u32)));
```

- the global memory case

```c
// Copy host memory to global memory section in GPU
CUDA_CALL(cudaMemcpyToSymbol(gmem_data_gpu, const_data_host,
    KERNEL_LOOP * sizeof(u32)));
```

For each case, launch the respective CUDA kernel
__global__ void const_test_gpu_const(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = const_data_gpu[0];

        for (int i=0; i<KERNEL_LOOP; i++)
        {
            d ^= const_data_gpu[i];
            d |= const_data_gpu[i];
            d &= const_data_gpu[i];
            d |= const_data_gpu[i];
        }

        data[tid] = d;
    }
}
__global__ void const_test_gpu_gmem(u32 * const data, const u32 num_elements)
{
    const u32 tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < num_elements)
    {
        u32 d = gmem_data_gpu[0];

        for (int i=0; i<KERNEL_LOOP; i++)
        {
            d ^= gmem_data_gpu[i];
            d |= gmem_data_gpu[i];
            d &= gmem_data_gpu[i];
            d |= gmem_data_gpu[i];
        }

        data[tid] = d;
    }
}