GPU PROGRAMMING CHALLENGES

SACHIN JOSHI | UDAY PITAMBARE
INDIANA UNIVERSITY BLOOMINGTON

Guided by: Prof. Arun Chauhan

Agenda

- Sample CUDA Program
- Programming challenges
- Optimization challenges
- Current research
- Questions

Sample CUDA Program

```
__global__ void kernel(int *array)
{
     int  x = /* Do some operation using threadIdx,  blockIdx */
     array[ some index] =  x;
}


int main(void)
{
  int num_elements =  256;
  int num_bytes =  num_elements * sizeof(int);

  int *device_array = 0;
  int *host_array = 0;

  host_array = (int*)malloc(num_bytes);
  cudaMalloc((void**)&device_array, num_bytes);  //linear mem on device (global mem)

  int block_size = 128;

  int grid_size = num_elements / block_size;  // no. of blocks

  kernel<<<grid_size, block_size>>>(device_array); // invoke kernel


  cudaMemcpy(host_array, device_array, num_bytes,    cudaMemcpyDeviceToHost);


free(host_array);
cudaFree(device_array);

}
```

Programming challenges

- Rewriting code
- Playing with memory
- Kernels
- Debugging

Rewriting code

- Did you wish there was just a compile time option ?

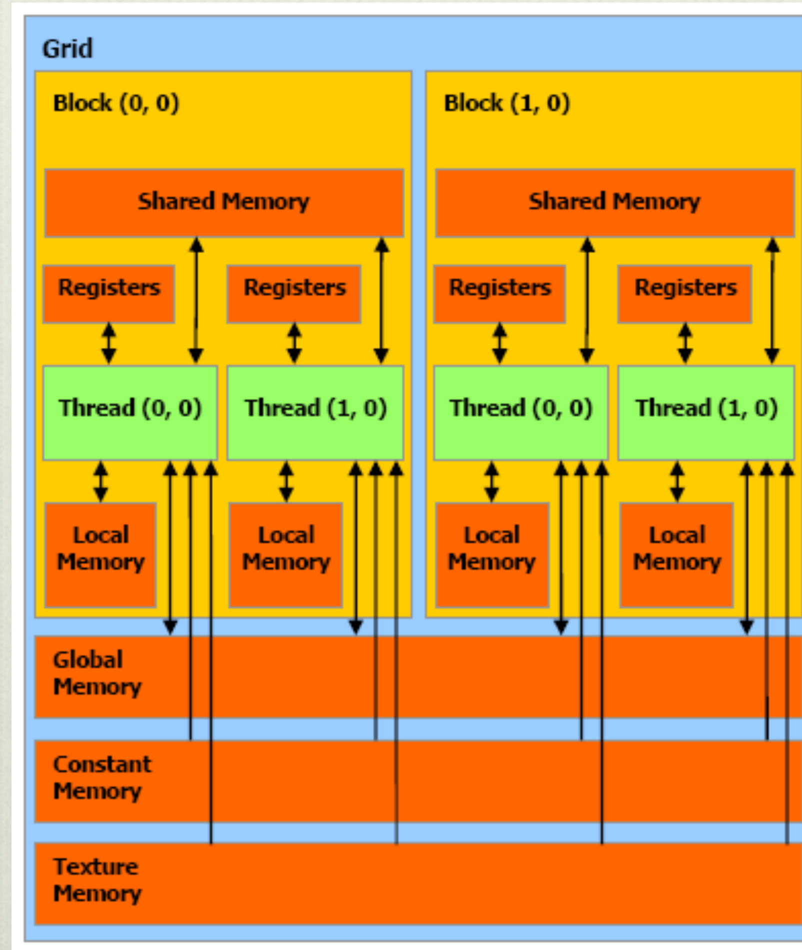
Compilers cannot magically compile your code that runs on a CPU to make it run on a GPU.

- A lot of code needs to re-written and code size increased by ‘n’ times.

Remember matrix multiplication ?

- OpenCL : Programmer has to write (nearly)same tedious boiler-plate code everytime.

Playing with memory



So where is my variable?

Variable declaration	Memory	Scope	Lifetime
automatic variables other than arrays	register	thread	kernel
automatic array variables	global/local	thread	kernel
__device__ __shared__ int nShareVar;	shared	block	kernel
__device__ int Globalvar;	global	grid	application
__device__ __constant__ int nConstvar;	constant	grid	application

Restrictions

- **__shared__** and **__constant__** cannot be used in combination with each other, obviously.
- **__constant__** variables cannot be assigned to from the device, only from the host. (cudaMemcpyToSymbol)
- Not possible to allocate memory on the fly inside __global__ or __device__ functions (1.x)

    Pass size as 3rd param in execution config

    Only one dynamically-sized shared memory array per kernel is supported.

    In your kernel code,

    ```
    extern __shared__ float myData[];
    ```

Know your limits (Tesla)

- Total number of cores : 30 x 8 = 240
- Global memory : 4GB
- Shared memory per SM : 16KB
- Number of 32-it SM/register : 8K/16K
- Constant memory : 64KB i.e 8KB/SM
- Max threads/block : 512
- Warp size : 32

Why should I know the limits?

- Insufficient space to hold variables, then offload to local memory. (Remember slow ?)

  (–maxrregcount=N)

- To determine blocksize. Usually a multiple of warpsize.

Kernels

- __global__ qualifier.
- Cannot call another kernel, host function or make recursive calls
- Not performance portable.
- Function calls inside kernel get inlined.
- Did you try passing double pointers in your matrix multiplication example?

  Wonder why it didn't work ?

2D array problem

- What was the missing part ?

```
// Copy the device pointer list first, you cannot access host mem from device code

int **d_a;
cudaMalloc((void ***)&d_a, N * sizeof(int *));
cudaMemcpy(d_a, h_a, N*sizeof(int *), cudaMemcpyHostToDevice);

int ** h_b = (int **)malloc(2 * sizeof(int *));

for(int i=0; i<N;i++)
    {
        cudaMalloc((void**)&h_b[i], 2*sizeof(int));
        cudaMemcpy(h_b[i], &bb[i][0], 2*sizeof(int), cudaMemcpyHostToDevice);
    }
```

- But in the end why would you want to do this when you can do the same thing using 1D array and which is more efficient.

Function Type Qualifiers restrictions

- __**device**__ functions are always inlined
- __**device**__ and __**global**__ do not support recursion
- ____**device**__ and __**global**__ functions cannot have a variable number of arguments.
- __**device**__ functions cannot have their address taken
- __**global**__ functions must have void return type

Debugging

- CUDA-GDB / VS Nsight
- Stepping is at the granularity of a warp. Cannot stop a particular thread
- Cannot step over a subroutine.

Other issues

- Divide-by-zero in GPU thread will not halt the program. Inf/Nan will persist and propogate.
- Do not mix host pointers and device pointers.
- Returning error codes from kernel to the host ?
- __shared__ int shared_var = threadIdx.x; What will happen?

OPTIMIZATION CHALLENGES

- Reducing global memory accesses
- Bank conflicts
- Control flow

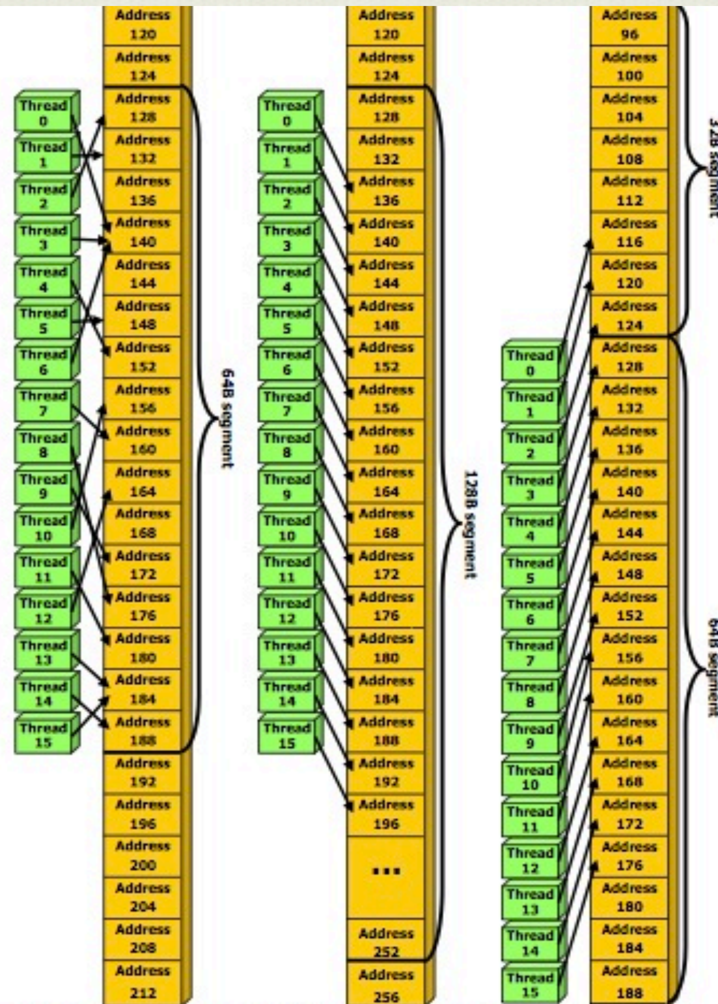
…Solutions

- Memory Coalescing
- Bank conflict avoiding techniques
- Remove ‘IFs'

Memory Coalescing

- If the threads in a block are accessing consecutive global memory locations, then all the accesses are combined into a single request by the hardware.
- This increases global memory bandwidth and instructions throughput.
- For compute capability < 1.2, addresses to be accessed must be located in contagious locations to achieve memory coalescing.
- Latest hardware (compute capability 1.2 and later) relaxes conditions to be satisfied for coalescing.

Memory Coalescing Contd…



Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.

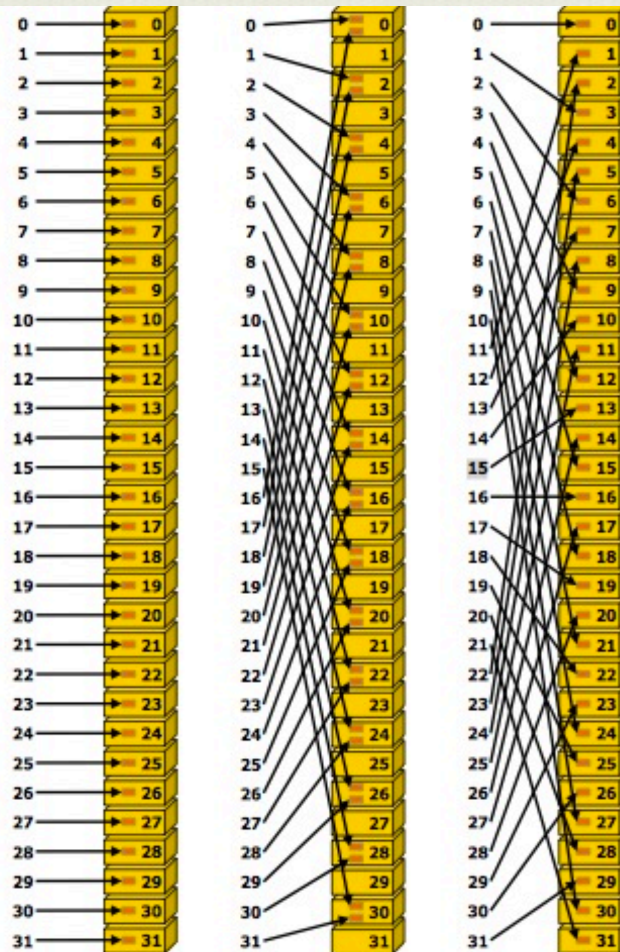
Memory Coalescing Contd…

- Use nearby addresses for threads in a warp
- Use unit stride wherever possible
- Structure of arrays

Memory Banks Conflict

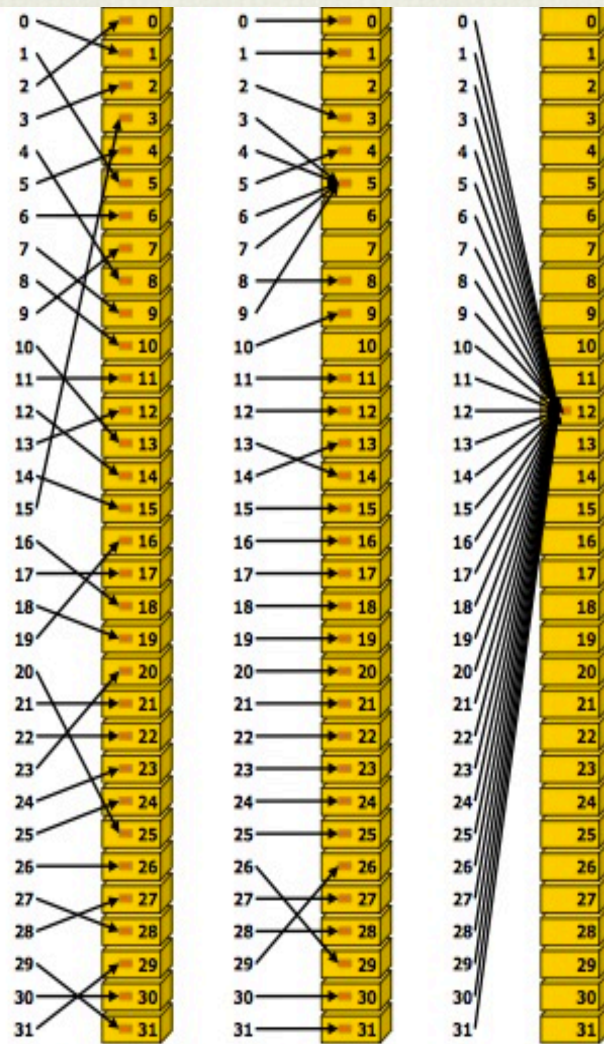
- To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called banks.
- Memory read/write made of n addresses in n distinct banks can be serviced simultaneously.
- Bank conflict is said to have occurred if two addresses of a memory request fall in the same memory bank and the access has to be serialized.

Bank Conflict Contd…



Left: Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Bank Conflict Contd…



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Avoiding Memory Bank Conflicts

- Memory padding

Control Flow

- Performance is hampered if there are two many ‘if’ statements in the program.
- ‘If’ is executed for every thread in the wrap, condition turns out to be true for only one or few threads.
- For other threads, comparisons are unnecessary.
- Solution: *avoid ‘ifs’ in kernel code.*

OTHER CHALLENGES

- No officials support in higher level programming languages.
- Must program GPU part in C-like language and host part in higher level language
- Kernels are not performance portable.
- Parallel Programming is hard.

ONGOING RESEARCH/ WORK

- Copperhead
  - Python-like data parallel language and compiler
  - Currently, just converts Python to CUDA
  - 3.6 times fewer lines of code than CUDA
  - 45-100% of the performance of hand-crafted, well optimized CUDA code.
- HARLAN
  - A declarative approach for GPGPU programming
  - Programmer specifies ‘what’ and not ‘how’
  - Especially promising for GPU/CPU hybrid clusters
- OPENMP to CUDA
- PGI Fortran to CUDA (Portland Group)

CONCLUSIONS

- Understand the parallel architecture
- Understand how application maps to architecture
- Minimize data transfer between host and device
- Enable global memory coalescing i.e. optimize memory access pattern.
- Avoid bank conflicts
- Take care of control flow (avoid warp divergence)

References

[1] *developer.download.**nvidia**.com*

[2] http://forums.nvidia.com

[3] 'Optimizing GPU Performance' by John Nickolls.

[4] 'Declarative Parallel Programming for GPUs' by Chauhan, Mahajan, Lumsdaine, Holk, Byrd, Willcock

[5] 'OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization' by Lee, Min, Eigenmann

[6]' Copperhead: A Python-like Data Parallel Language & Compiler' by Catanzaro, Garland, Keutzer.

QUESTIONS?