Preparing seismic codes for GPUs and other many-core architectures

Paulius Micikevicius

Developer Technology Engineer, NVIDIA

2010 SEG Post-convention Workshop (W-3)
High Performance Implementations of Geophysical Applications
Oct 21, 2010

Outline

- **High level review of various GPUs**
- **Common features to all GPUs**
- **Requirements for maximizing GPU performance**
- **Requirements as applied to**
  - Kirchhoff Migration
  - Reverse Time Migration

Terminology

- **We'll define a "brick" for this talk**
  - Since all vendors use different names
  - GPUs are built by replicating bricks (10s)
    - Connected to memory via some network
- **Brick = minimal building block that contains own:**
  - Control unit(s) - decodes/issues instructions
  - Registers
  - Pipelines for instruction execution
  - Local cache(s)

NVIDIA GPU Brick (Streaming Multiprocessor)

- **Up to 1536 threads**
- **Instructions are issued per 32 threads (*warp*)**
  - Think 32-way vector of threads
- **Source code is for a single thread and is <u>scalar</u>:**
  - No vector intrinsics a la SSE
  - HW handles grouping of threads into vectors, vector control flow
- **Dual-issue: instructions from different warps**
- **Shared memory, L1 cache**
- **Large register file, partitioned among threads**

AMD GPU Brick (SIMD Engine)

- **Up to ~1500 threads**
- **Instructions are issued per 64 threads**
- **VLIW instruction issue:**
  - HW designed for 5 “issue” slots (16x5 ‘cores’ per brick)
  - Combine up to 5 instructions from the same thread to maximize performance
- **Source code is for a single thread and is <u>scalar</u>:**
  - HW handles grouping threads into vectors and control flow
  - Compiler handles VLIW combining
- **Shared memory, L1 cache**
- **Large register file, partitioned among threads**

Intel Larrabee/Knights Ferry/Corner Brick (Core)

- **Up to 4 threads**
- **Scalar and vector (512-bit SIMD) units**
  - For example: 16-fp32 vector SIMD
- **Dual issue: scalar-vector, from the same thread**
- **Source code is for a single thread and is <u>vector</u>:**
  - Intrinsics for SIMD operations (a la SSE)
- **L1 and L2 caches**
  - Intrinsics for pre-fetching and prioritization of cache lines
  - No user-managed shared memory
- **Small register file (relies on caches)**

HW Commonalities

- **Built by replicating 10s of "bricks"**
  - In-order instruction issue
- **High GPU memory bandwidth (150+ GB/s)**
- **"Bricks" are vector processors**
  - Different execution paths within vectors are supported but degrade performance
  - Different execution paths in different vectors have no impact on performance
- **Vectors access memory in cache-lines**
  - Consecutive threads (vector elements) should access a contiguous memory region
  - Scattered access is supported, but will fetch multiple lines, increasing bandwidth-pressure

Requirements for Maximum Performance

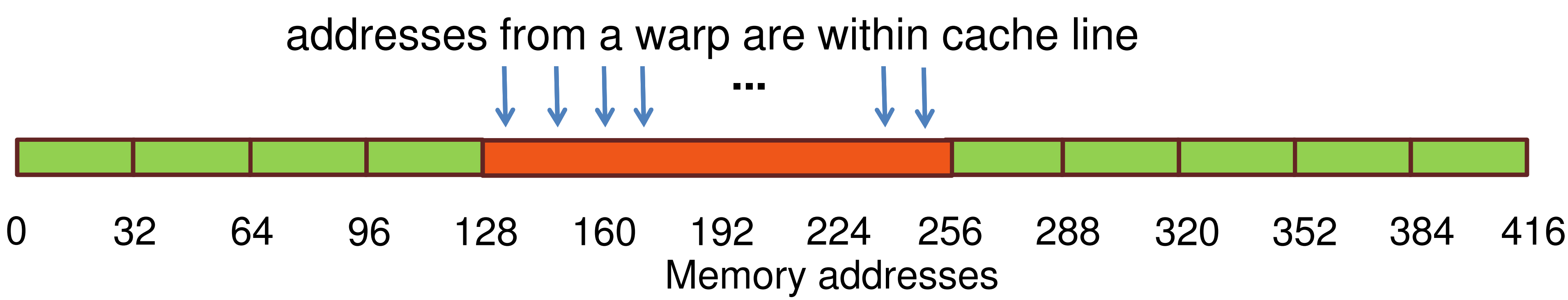
- **Have sufficient parallelism**
  - At least a few 1,000 of threads per function
- **Coherent memory access**
  - By threads in the same “thread-vector”
- **Coherent execution**
  - By threads in the same “thread-vector”

Amount of Parallelism

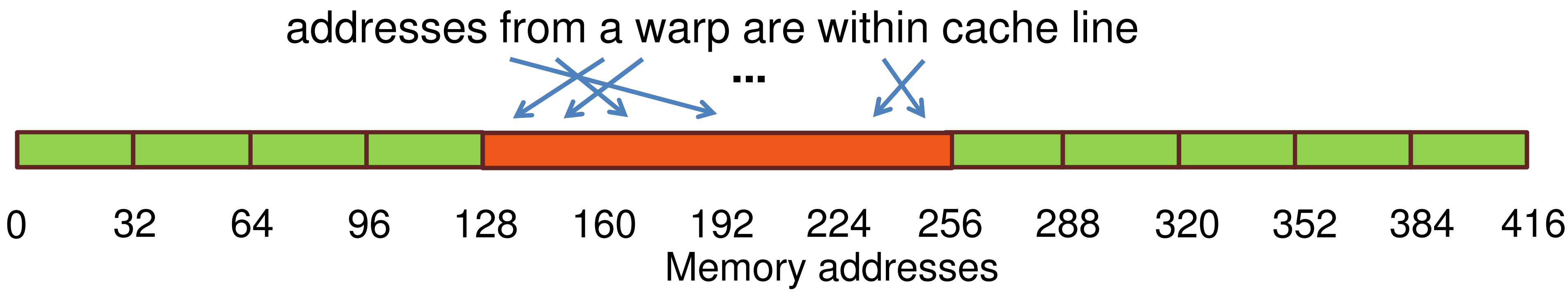
- **GPUs issue instructions in order**
  - Issue stalls when instruction arguments are not ready
- **GPUs switch between threads to hide latency**
  - Context switch is free: thread state is partitioned (large register file)
- **Conclusion: need enough threads to hide math latency and to saturate the memory bus**
  - Independent instructions within a thread also help
- **Very rough rule of thumb:**
  - Need ~512 threads per “brick”
  - So, at least a few 1,000 threads per GPU

Memory Access

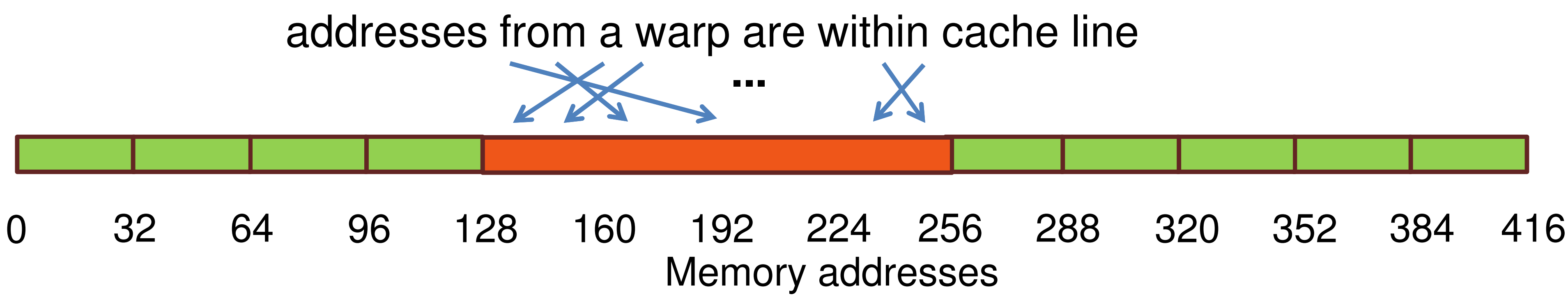
- **Addresses from a warp ("thread-vector") are converted into line requests**
  - NVIDIA line sizes: 32B and 128B
  - Goal is to maximally utilize the bytes in these lines

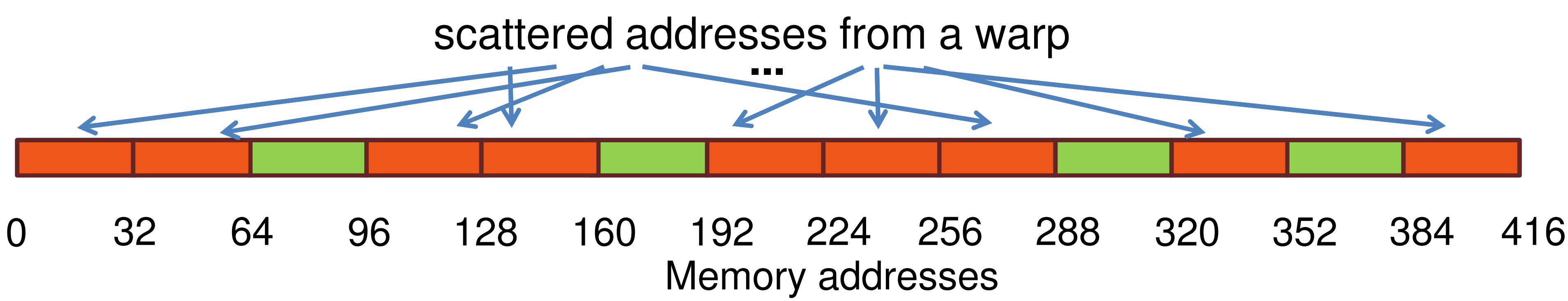


Memory Access

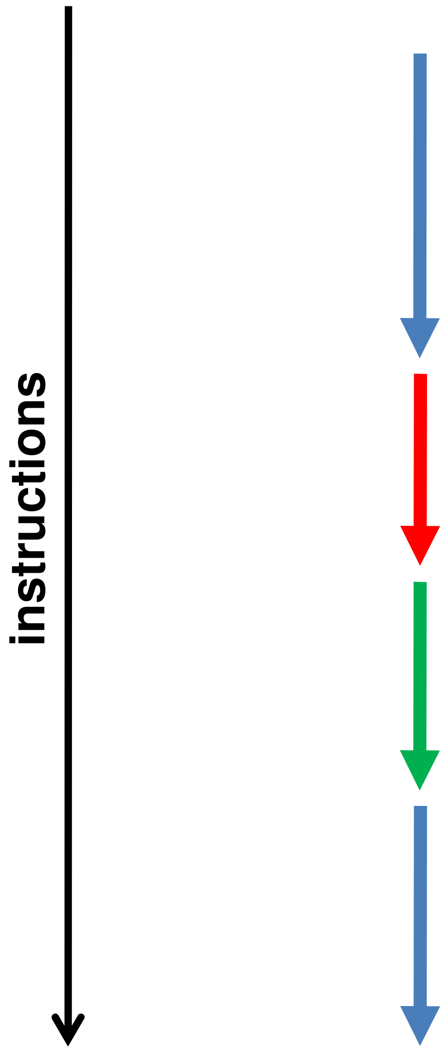


Memory Access



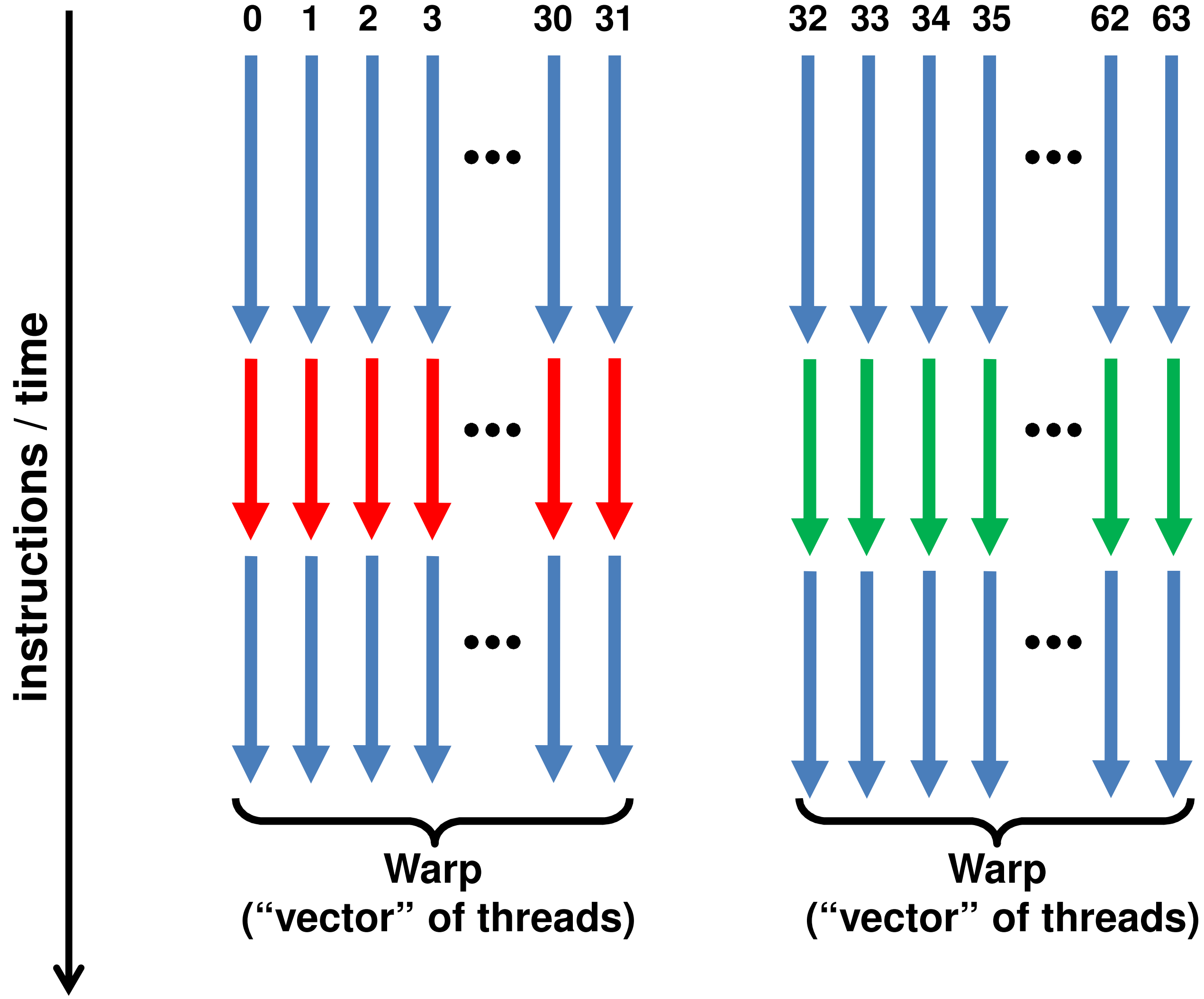


Coherent Execution

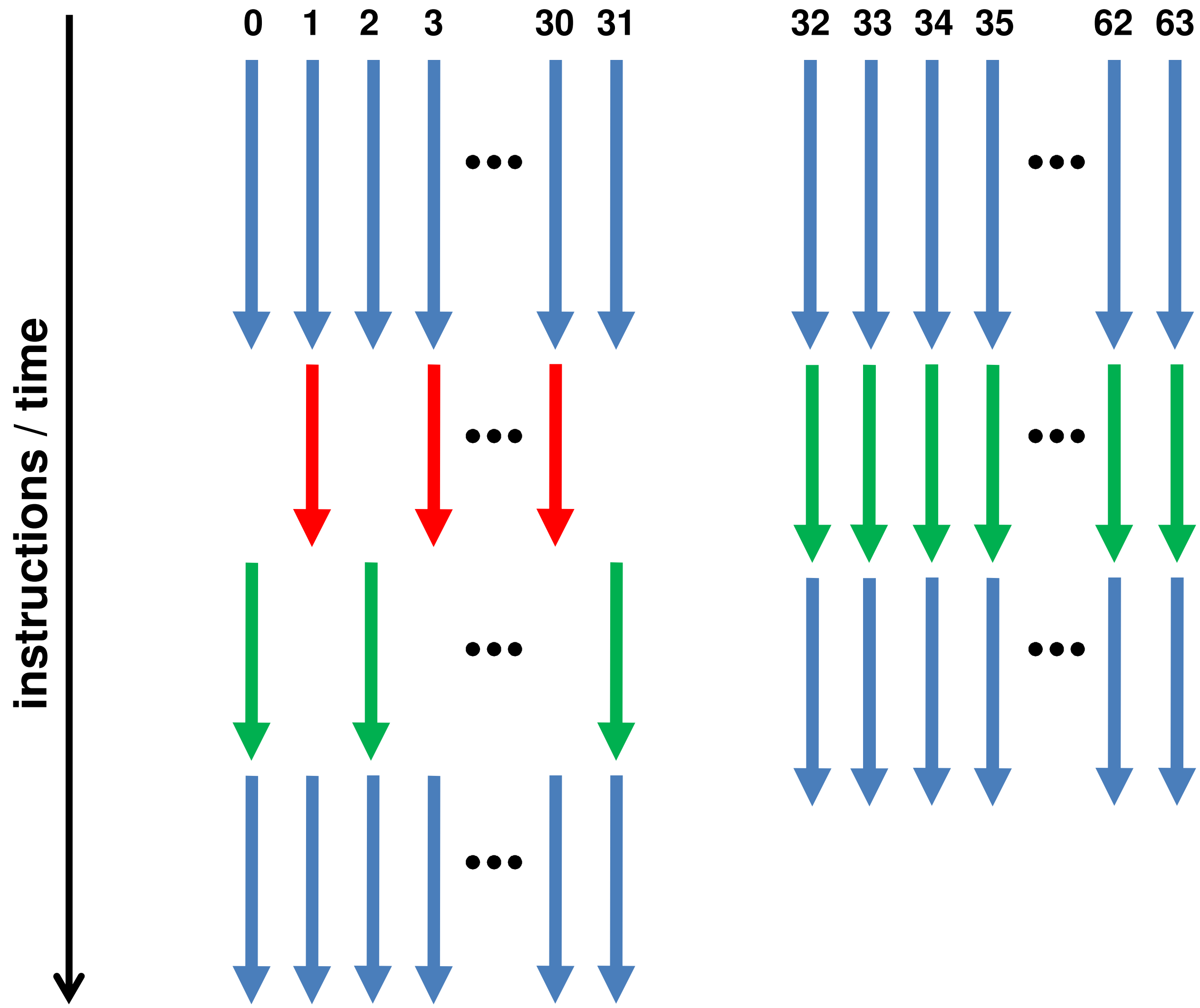



```
if ( … )
{
    // then-clause
}
else
{
    // else-clause
}
```

Execution within warps is coherent



Execution diverges within a warp



Requirements for Maximum Performance

- **Have sufficient parallelism**
- **Coherent memory access**
- **Coherent execution**

Kirchhoff Migration

One (very) simplified way to look at it:

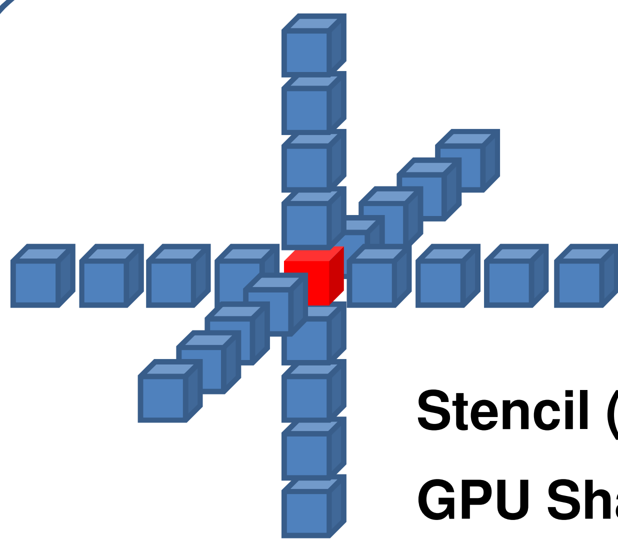
```
for each input trace (src-rcv pair) do
  for each output point do
    compute travel-times
    get input trace value(s) based on travel times
    update the output point
```

Kirchhoff Migration

- **Amount of parallelism**
  - Plenty of output points
  - For example: 1 or several points per thread
- **Coherent memory access**
  - Successive threads should process adjacent output points
    - Fastest varying dimension for thread IDs = fastest varying dimension for data
    - Coherent writes
    - Reads will be scattered, but within close proximity
- **Coherent execution**
  - Usually all threads will execute the same code path

Reverse Time Migration

- **The main component is FD computation**
  - Here I assume we work in time domain
- **Derivatives are computed with stencils**



Stencil (8th order in space)

GPU Shared Memory is utilized to reuse input points among neighboring threads

Reverse Time Migration

- The main component is FD computation
- Derivatives are computed with stencils

```
for each FD step do
    for each output point do
        compute wave-field derivatives
        combine derivatives with property-fields
        update the output point
```

Reverse Time Migration

- **Amount of parallelism**
  - Plenty of output points (100s of millions)
  - For example: 1 or several points per thread
- **Coherent memory access**
  - Successive threads in a warp should process adjacent points
    - Fastest varying dimension for thread IDs = fastest varying dimension for data
    - 2D blocks of threads -> exploit wavefield locality from SMEM
    - Coherent writes and reads
      - Regular grid -> regular reads as well
- **Coherent execution**
  - All threads execute the same code path
    - Execution is not data dependent

Summary of Coding Patterns for GPUs

- **Seismic problems have plenty of parallelism**
  - Output points, traces/shots
  - Need to make sure that the framework exposes that when calling functions parallelizable on accelerators
- **Memory accesses can be made coherent enough**
  - Access patterns are often coherent, or scattered in reasonably close proximity (for L1 or tex cache on GPU)
  - Generally, make sure that thread-ID varying-order matches data varying-order
    - If arranging threads into 2D blocks, fastest varying dimension should be accessing in multiples of cache lines
- **Strive for coherent execution**
  - Preferably group together threads (hence data) that follows the same path

Comparing GPU and CPU requirements

- **Amount of parallelism**
  - CPUs need less currently, but requirement is growing:
    - Increasing core count
    - Increasing vector width
- **Coherent memory access**
  - Already needed to maximize SSE throughput
  - Scattered accesses also underutilize bus bandwidth
- **Coherent execution**
  - Already needed within SSE vectors

References

- **NVIDIA:**
  - http://www.nvidia.com/object/fermi_architecture.html
  - Fermi Compute Architecture Whitepaper (http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf)
- **AMD:**
  - Unleashing the Power of Parallel Compujte with Commodity ATI Radeon 5800 GPU, Siggraph Asia 2009 (http://sa09.idav.ucdavis.edu/docs/SA09_AMD_IHV.pdf)
- **Intel:**
  - L. Seiler et al. Larrabee: A Many Core X86 Architecture for Visual Computing. Siggraph 2008 (http://software.intel.com/file/18198/)
  - Tom Forsyth. The Challenge of Larrabee as GPU. Colloquium at Stanford, 2010 (http://www.stanford.edu/class/ee380/Abstracts/100106.html)

References

- **“Seismic Imaging” by S. Morton**
  - http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_Seismic_Hess.pdf
- **“Accelerating Kirchhoff Migration by CPU and GPU Cooperation”, J. Panetta *et al***
  - http://www.cos.ufrj.br/~monnerat/SBAC_2009.html
- **“Implementing 3D Finite Difference Codes on the GPU” by P. Micikevicius**
  - Slides: http://www.nvidia.com/content/GTC/documents/1006_GTC09.pdf
  - video (slides+audio): http://developer.download.nvidia.com/compute/cuda/docs/GTC09Materials.htm then look for talk 1006

BACKUP

Current Differences among GPUs

- **Hardware:**
  - Vector width: 16, 32, 64
  - VLIW (AMD) vs dual-issue (NVIDIA, Intel)
  - Dual-issue: same thread (Intel) vs different threads (NVIDIA)
  - Large register file, small cache (NVIDIA,AMD) vs small register file, larger caches (Intel)
- **Programming model**
  - Intel: vector source code (SIMD intrinsics)
  - AMD,NVIDIA: scalar source code
    - hw aggregates threads into vectors and resolves control flow divergence

Simple 2D Stencil Code in CUDA (OpenCL equivalent in comments)

```
__global__ void stencil_2d( float *output, float *input,
                            const int dimx, const int dimy, const int row_size )
{
   int ix = blockIdx.x*blockDim.x + threadIdx.x;  // ix = get_global_id(0);
   int iy = blockIdx.y*blockDim.y + threadIdx.y;  // iy = get_global_id(1);
   int idx = iy * row_size + ix;

   output[idx] = -4 * input[idx]
                 + input[idx+1]  + input[idx-1]
                 + input[idx + row_size] + input[idx – row_size];
}
```