A Study of Productivity and Performance of Modern Vector Processors

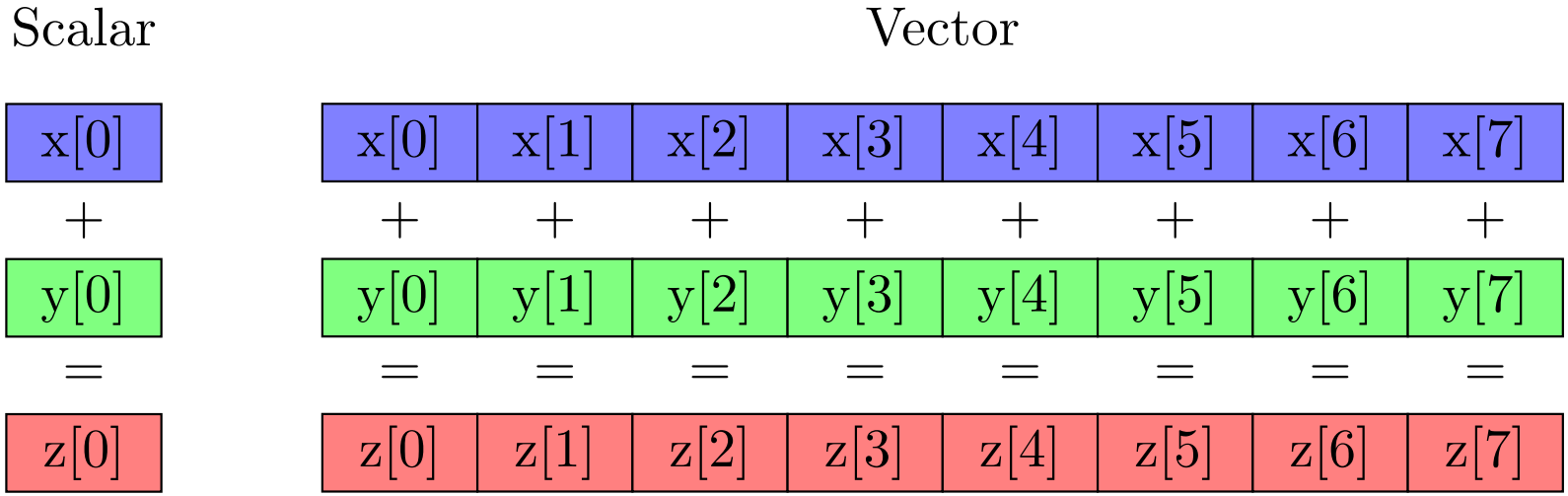
Paul Springer

29.03.2012

Contents

Vector Processors

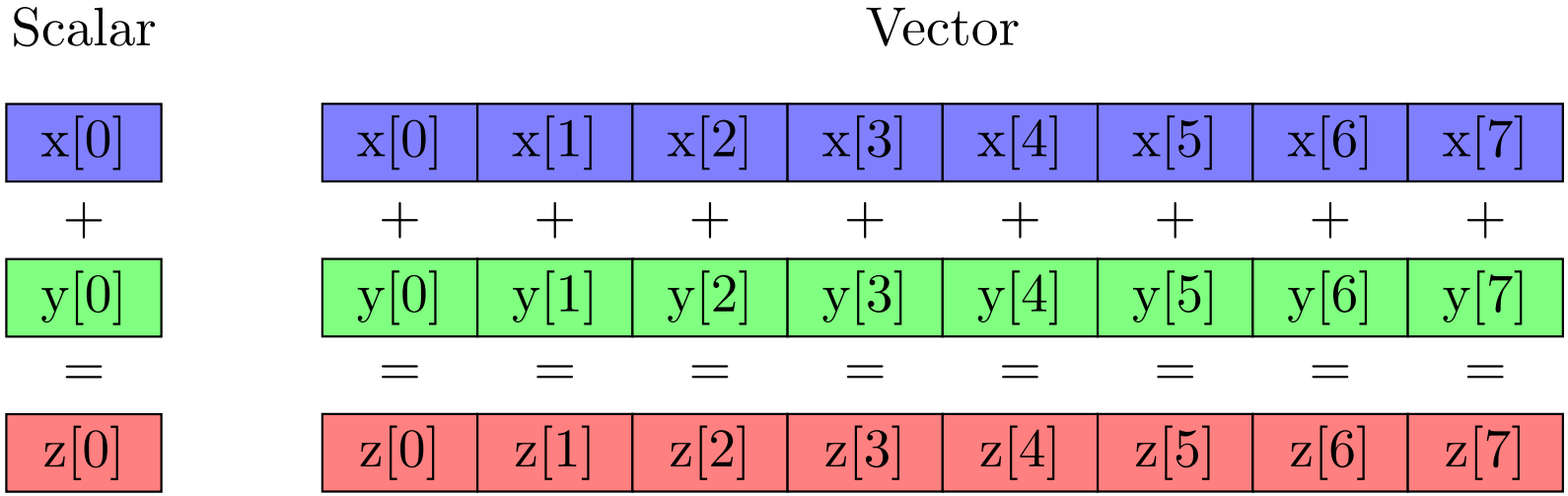
- Operate on multiple data elements in lock-step
- Theoretical speedup depends on vector-width



- Large vector-width = high speedup?

Vector Processors

- Operate on multiple data elements in lock-step
- Theoretical speedup depends on vector-width



- Large vector-width = high speedup?

Vector Processors

- Intel's Xeon E5-2670 CPU
  - 8 cores at 2.6 GHz each
  - Process 8 SP/ 4 DP elements in lock-step
  - Peak SP performance of 166.4 GFLOPS
  - Peak DP performance of 83.2 GFLOPS
- NVIDIA's Quadro 6000 GPU
  - 448 cores at 1.15 GHz each
  - Process 32 SP/ 16 DP elements in lock-step
  - Peak SP performance of 1030.4 GFLOPS
  - Peak DP performance of 515.2 GFLOPS

Magnetoencephalography (MEG)

- Non-invasive imaging technique
- Display the neuronal activity within the human brain [3]
- Measure the magnetic field outside the human head
- Neuromagnetic inverse problem

Magnetoencephalography (MEG)

- Formulated as an minimization problem [1]

$$\text{Find } x \in \mathbb{R}^k \text{ such that } f(x) \longrightarrow min \tag{1}$$

- Objective function $f : \mathbb{R}^k \longrightarrow \mathbb{R}$

$$f(x) := \left\| \begin{bmatrix} L \\ \lambda I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_p^p \tag{2}$$

- $L \in \mathbb{R}^{n \times k}$, dense
- $x \in \mathbb{R}^k$, dense
- $b \in \mathbb{R}^n$, dense
- I is the $k \times k$ identity, with $n \ll k$

MEG Software Package

- Introduced by Bücker et al. [2]
- First- and second-order derivatives for faster convergence
- Divided into three computational routines
  - *Eval*: Evaluate $f(x) \in \mathbb{R}$
    - Matrix-vector product
  - *Gradient*: Evaluate $f(x)$ and $z = \nabla f(x) \in \mathbb{R}^k$
    - Matrix-vector product
    - Transposed matrix-vector product
  - *Hessian*: Evaluate the Hessian-vector product $(\nabla^2 f(x))y \in \mathbb{R}^k$
    - Two matrix-vector products
    - Transposed matrix-vector product

Central Processing Unit (CPU)

OpenMP + auto-vectorization and Intel OpenCL

Sandy Bridge Architecture

- Multi-core architecture
- Non-Unified Memory Access (NUMA)
- Advanced Vector Extension (AVX)
  - Single Instruction Multiple Data (SIMD)
  - 256-bit vector width
  - Theoretical speedup of 8x (SP) and 4x (DP)
- Full cache-hierarchy
  - L1 and L2 cache dedicated to each core
  - L3 cache dedicated to each socket (shared among cores)

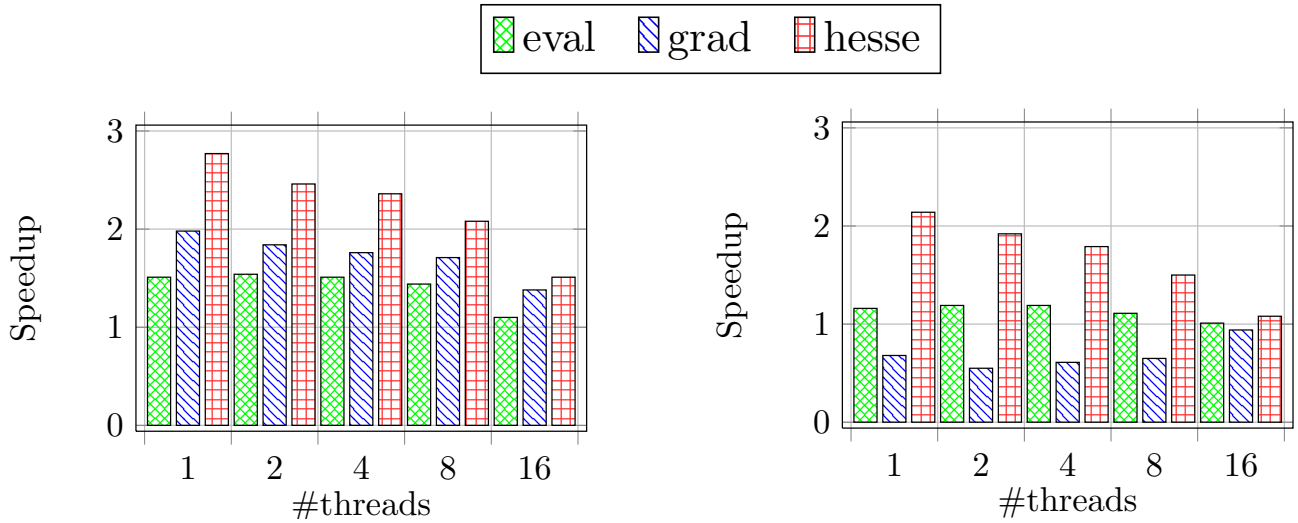
OpenMP

- Thread-level parallelism across the cores
- Data-level parallelism (SIMD) on each core
- Parallel initialization of the data
  - Pays attention to NUMA
  - First Touch Policy (FTP)
- Blocked algorithm
  - Cache-aware
- Utilize the auto-vectorization capabilities of the compiler
  - Ideally no further programming effort required
  - In reality code reorganization is often necessary

Programming Guidelines - Auto-Vectorization

- Avoid divergence based on the iteration
- Avoid non-contiguous memory accesses
- Avoid indirect memory accesses
- Use single entry, single exit loops
- ... and many more [4]

Performance Results



(a) Single precision (b) Double precision

Figure: Speedup due to vectorization. Threads are pinned.

- Speedup decreases with increasing number of threads
- Almost no speedup for 16 threads for DP
- DP benefits less than SP
- *Hessian* routine benefits the most

Performance Results

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
GFLOPS	33.3	37.8	43.6	14.3	14.1	22
Speedup FTP	2.1	1.8	1.8	2.0	1.9	1.9
Speedup C	16.3	15.3	15.7	13.9	13.5	15.8

Table: Final OpenMP results using 16 threads, running on two Intel Xeon E5-2670 CPUs.

- *Hessian* routine achieves best performance
- SP performance $\approx 2\times$ higher than DP performance
- FTP has a significant impact on performance
- Almost linear speedup
- *Blocked* version $\approx 2\times$ faster than *non-blocked* version

Summary OpenMP

- Vectorization requires additional effort
- Vectorization-Speedup decreased with increasing #threads
- Little speedup due to vectorization
- Performance is sensitive to the underlying architecture
  - Caching
  - NUMA

Introduction to the Open Computing Language (OpenCL)

OpenCL

- Programming heterogeneous systems (e.g. CPUs + GPUs)
- Single Program Multiple Data (SPMD)
  - The program (*kernel*) is executed multiple times
  - A running instance of a kernel is called *work-item*
- Execution Model
  - Index Space
  - Work-group
  - Work-item

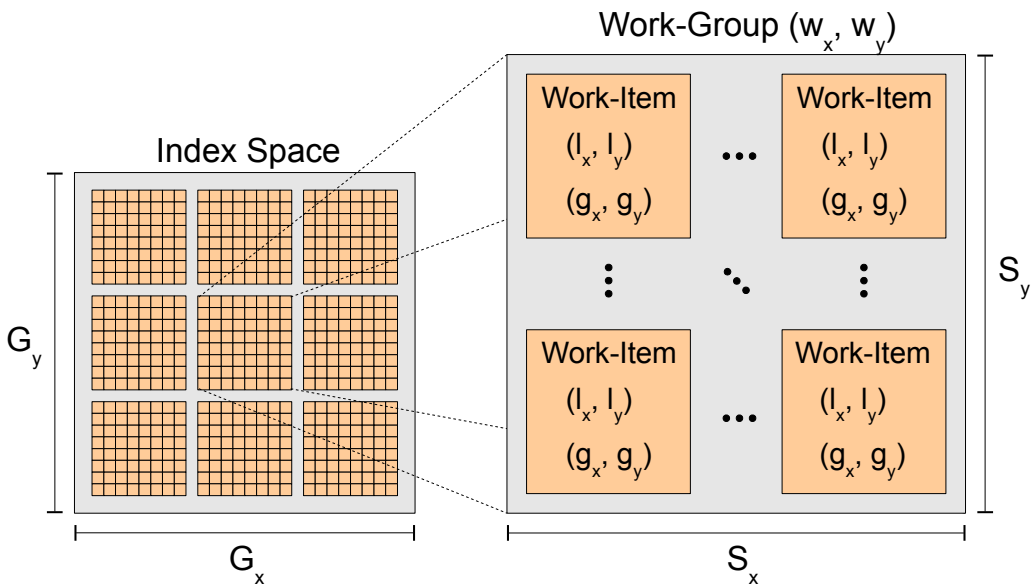


Figure: OpenCL Execution Model.

OpenCL

- *Host*
  - Interacts with the OpenCL context
- *Compute Device* (CD)
  - Executes commands
  - Can have multiple command-queues
- Platform Model
  - Compute Device
  - Compute Unit (CU)
  - Processing Element (PE)

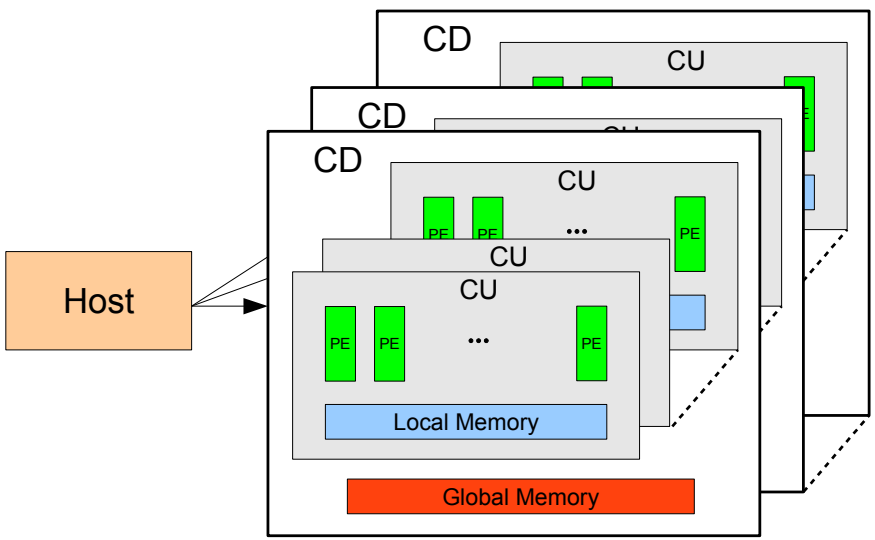


Figure: OpenCL Platform Model.

OpenCL

- *Host*
  - Interacts with the OpenCL context
- *Compute Device* (CD)
  - Executes commands
  - Can have multiple command-queues
- Platform Model
  - Compute Device
  - Compute Unit (CU)
  - Processing Element (PE)

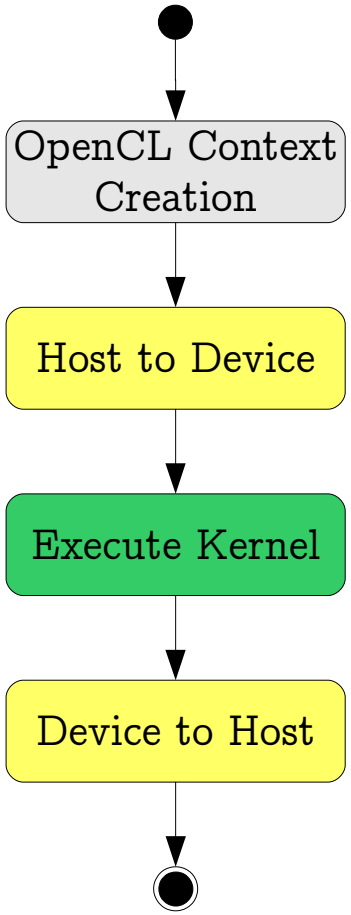


Figure: OpenCL activity diagram.

Intel OpenCL

Intel OpenCL - Mapping

- Multiple multi-core CPUs constitute a single CD
- Each core constitutes a single CU
- Each AVX lane corresponds to a single PE
- Main memory corresponds to global memory

Programming Guidelines

- Avoid memory transfers between host and device
- Launch at least as many work-groups as cores
- Tune the work-group size
- ... and many more [5].

Portability

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
Speedup	3.5	5.8	4.6	2.4	3.1	2.7

Table: Speedup of the tuned Intel OpenCL versions over the native GPU OpenCL versions. Both versions run on two Intel Xeon E5-2670 CPUs.

- OpenCL versions benefited from special tuning for the CPU

Performance Results

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
GFLOPS	31.0	27.3	30.5	13.9	12.7	15.6
Speedup OMP	0.9	1.0	0.7	1.0	0.9	0.7

Table: Final Intel OpenCL results. Run on two Intel Xeon E5-2670 CPUs.

- OpenCL versions yield almost the same performance as OpenMP versions
- SP performance $\approx 2\times$ higher than DP performance

Graphics Processing Unit (GPU)

Compute Unified Device Architecture (CUDA) and OpenCL

Fermi Architecture

- 14-16 CUs
- Up to 6 GB global memory
  - High latency
  - High throughput
- 48/16 KB local memory per CU
  - Low latency
  - High throughput
  - Divided into 32 banks
- Up to 1536 work-items per CU
  - High thread-level parallelism
  - Hide memory latency

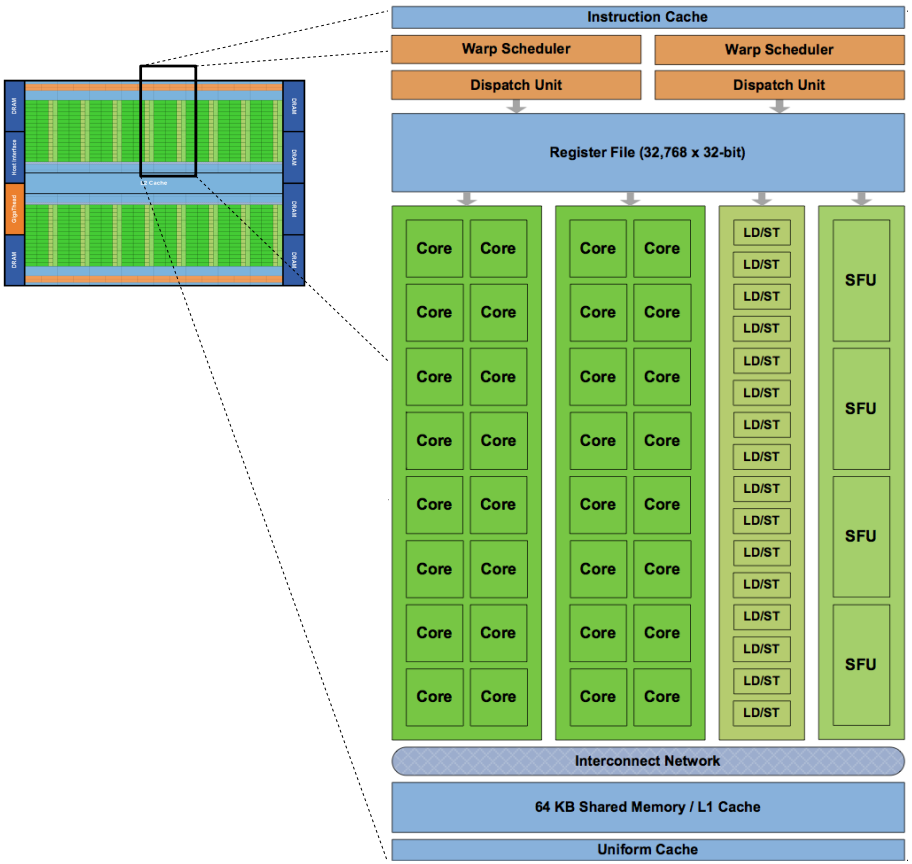


Figure: Fermi architecture [6].

Execution Model

- Work-groups
  - Divided into multiple warps
  - Share resources of the CU
- Warp
  - 32 consecutive work-items
  - Operate in lock-step
  - Single Instruction Multiple Thread (SIMT)
  - Issue global-memory accesses
  - Issue local-memory accesses

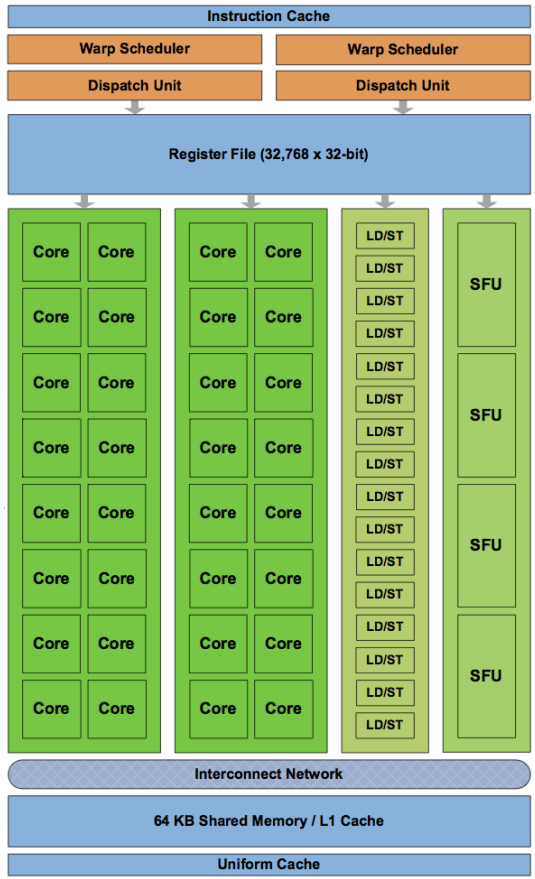
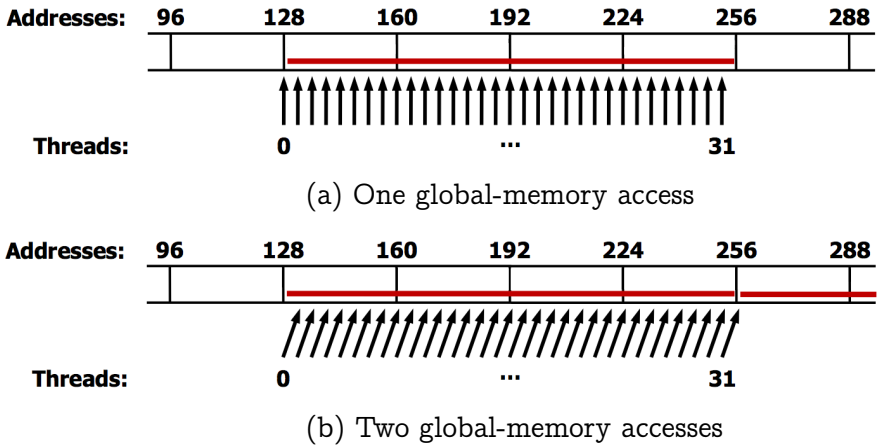


Figure: Fermi CU [6].

Programming Guidelines

- Coalesce memory accesses
- Avoid bank conflicts
- Reuse data (local memory)
- Hide memory latencies
- Avoid execution-path divergence within a warp
- ... many more [7]



(a) One global-memory access

(b) Two global-memory accesses

Figure: Global-memory access coalescing.

Programming Guidelines

- Coalesce memory accesses
- Avoid bank conflicts
- Reuse data (local memory)
- Hide memory latencies
- Avoid execution-path divergence within a warp
- ... many more [7]

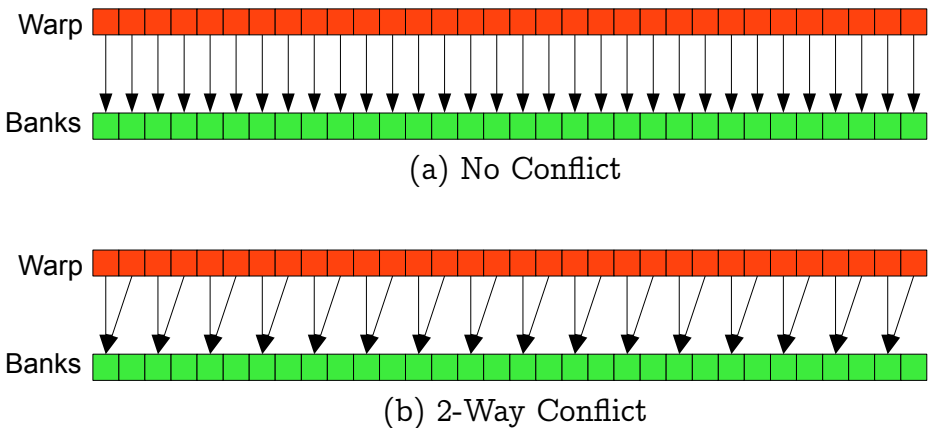


Figure: Local memory bank conflicts.

Programming Guidelines

- Coalesce memory accesses
- Avoid bank conflicts
- Reuse data (local memory)
- Hide memory latencies
- Avoid execution-path divergence within a warp
- ... many more [7]

Programming Guidelines

- Coalesce memory accesses
- Avoid bank conflicts
- Reuse data (local memory)
- Hide memory latencies
- Avoid execution-path divergence within a warp
- ... many more [7]

Programming Guidelines

- Coalesce memory accesses
- Avoid bank conflicts
- Reuse data (local memory)
- Hide memory latencies
- Avoid execution-path divergence within a warp
- ... many more [7]

Gradient Routine

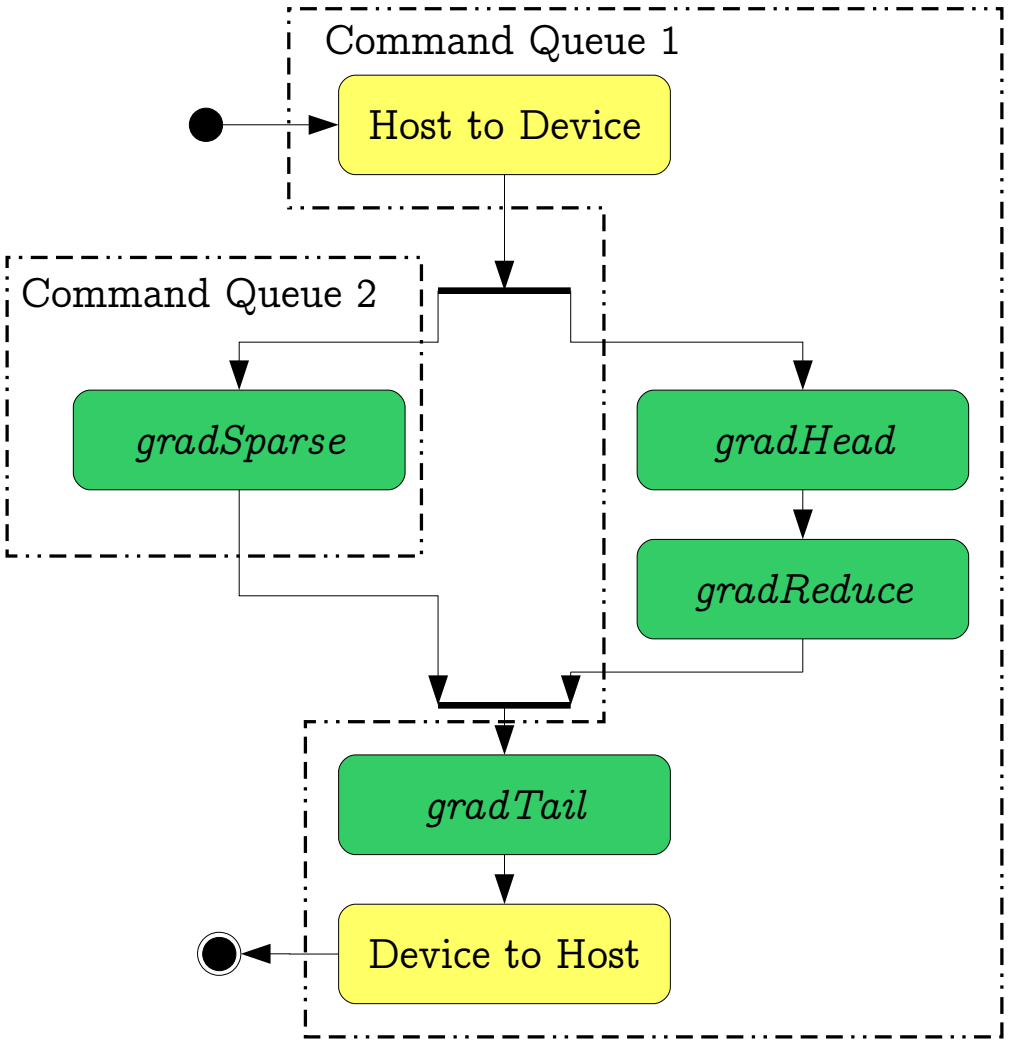
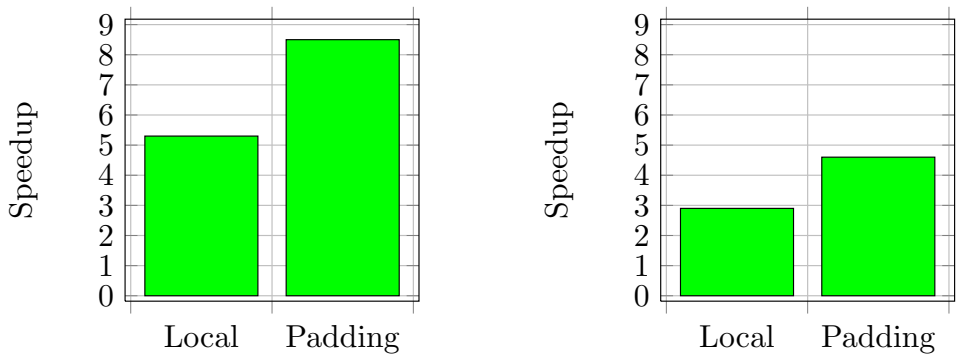


Figure: Activity diagram of the device for the *gradient* routine.

Performance Results



(a) Single precision (b) Double precision

Figure: Kernel *hesseHead* using CUDA. Speedup due to *local-memory usage* and local-memory usage in combination with *padding* (higher is better).

- Speedup due to local-memory usage is significant
- Resolved bank conflicts increase the performance by $\approx 60\%$
- DP benefits less from local memory than SP

Performance Results

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
CUDA	27.7	31.1	45.6	12.6	15.0	22.2
OpenCL	27.2	28.7	40.1	13.6	14.9	20.2

Table: GFLOPS of CUDA and OpenCL. Run on NVIDIA's Quadro 6000, including data transfers.

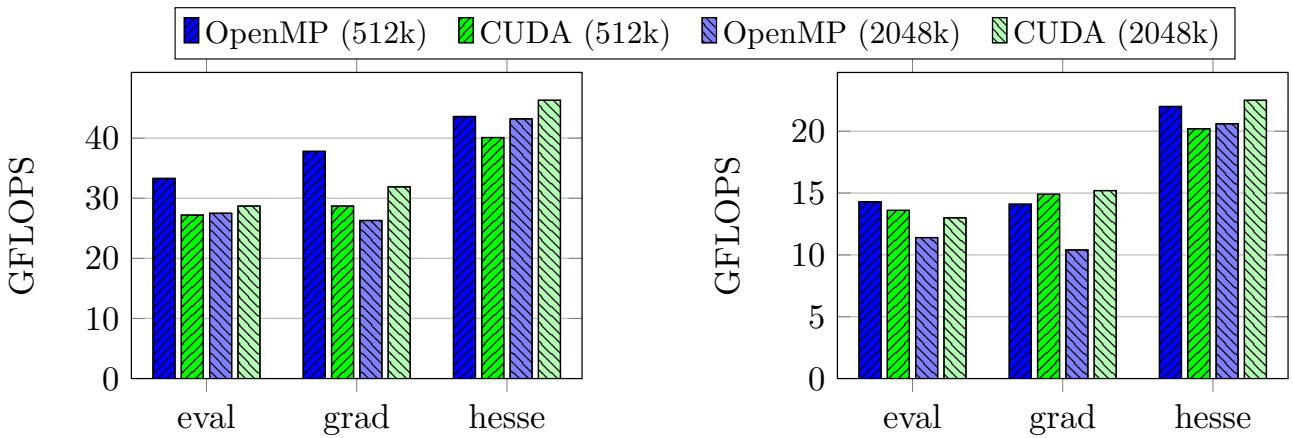
- Data transfers account for 10-20% of the runtime
- CUDA and OpenCL yield the same performance
- SP performance $\approx 2\times$ higher than DP performance
- Loop unrolling results in a significant speedup (up to $3.6\times$)

Summary CUDA/OpenCL

- Performance is sensitive to the underlying architecture
  - Local-memory usage
  - Register usage
  - Bank conflicts
  - Coalescing requirements
  - ... many more
- GPU algorithm more complicated than OpenMP counterpart

Comparison CPUs and GPUs

Performance Results



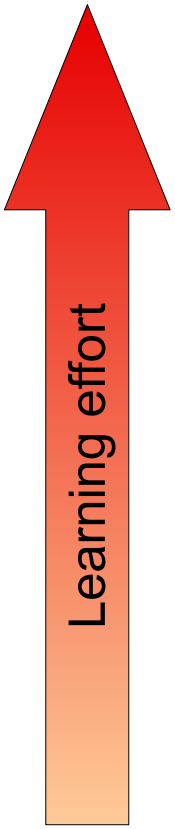
(a) Single precision (b) Double precision

Figure: GFLOPS (higher is better) of the routines using either CUDA or OpenMP.

- CPU and GPU roughly yield the same performance
- Smaller problem-size favours the CPU
- Larger problem-size favours the GPU

Learning Effort

1. NVIDIA CUDA/ OpenCL
   - Many architectural features
2. Intel OpenCL
   - OpenCL
   - Vectorization
3. Intel intrinsics AVX
4. OpenMP
   - Auto-vectorization



Source Lines of Code (SLOCs)

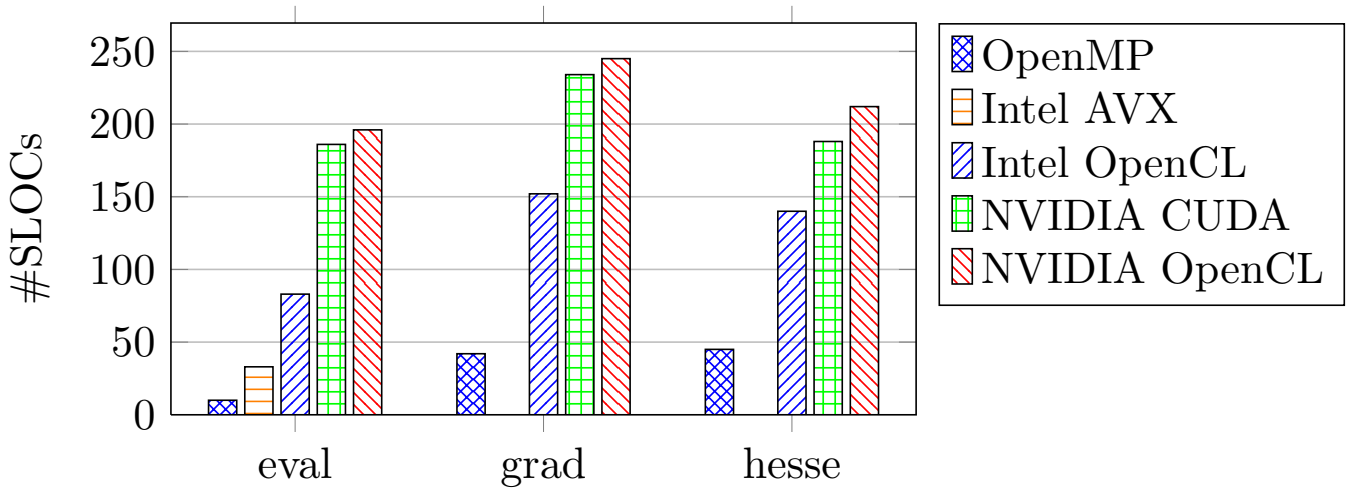
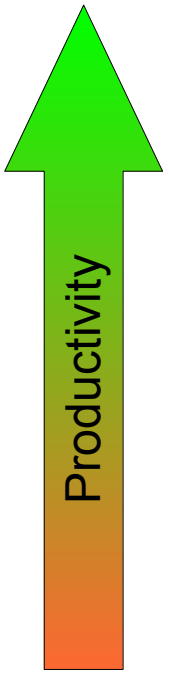


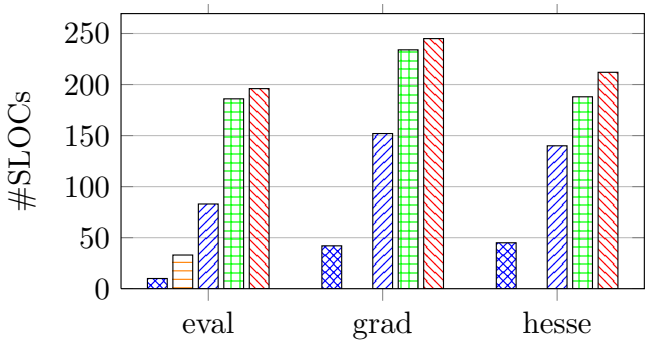
Figure: Code expansion. Added or modified SLOCs w.r.t. the C versions (lower is better). SLOCs of C version: 30(eval), 50(grad) and 45(hesse). Neglecting OpenCL context creation.

- OpenCL context creation can be reused
- GPU versions require additional data transfers

Productivity

1. OpenMP
2. Intel intrinsics AVX
3. NVIDIA CUDA/ OpenCL
4. Intel OpenCL





Conclusion

- Small speedup due to vectorization (CPU)
- CPU and GPU ...
  - ... achieved comparable performance
  - ... performance is sensitive to the underlying hardware
- GPU versions required more effort than their CPU counterparts
- Recommend OpenMP for the MEG application
- What might come...
  - ... increasing number of cores (e.g. NVIDIA's *Kepler*)
  - ... increasing SIMD support (e.g. Intel's *MIC*)
  - ... more powerful compilers (auto-vectorization/ -parallelization)
  - ... directive-based programming paradigms for GPUs

Conclusion

- Small speedup due to vectorization (CPU)
- CPU and GPU ...
  - ... achieved comparable performance
  - ... performance is sensitive to the underlying hardware
- GPU versions required more effort than their CPU counterparts
- Recommend OpenMP for the MEG application
- What might come...
  - ... increasing number of cores (e.g. NVIDIA's *Kepler*)
  - ... increasing SIMD support (e.g. Intel's *MIC*)
  - ... more powerful compilers (auto-vectorization/ -parallelization)
  - ... directive-based programming paradigms for GPUs

Conclusion

- Small speedup due to vectorization (CPU)
- CPU and GPU ...
  - ... achieved comparable performance
  - ... performance is sensitive to the underlying hardware
- GPU versions required more effort than their CPU counterparts
- Recommend OpenMP for the MEG application
- What might come...
  - ... increasing number of cores (e.g. NVIDIA's *Kepler*)
  - ... increasing SIMD support (e.g. Intel's *MIC*)
  - ... more powerful compilers (auto-vectorization/ -parallelization)
  - ... directive-based programming paradigms for GPUs

Conclusion

- Small speedup due to vectorization (CPU)
- CPU and GPU ...
  - ... achieved comparable performance
  - ... performance is sensitive to the underlying hardware
- GPU versions required more effort than their CPU counterparts
- Recommend OpenMP for the MEG application
- What might come...
  - ... increasing number of cores (e.g. NVIDIA's *Kepler*)
  - ... increasing SIMD support (e.g. Intel's *MIC*)
  - ... more powerful compilers (auto-vectorization/ -parallelization)
  - ... directive-based programming paradigms for GPUs

Conclusion

- Small speedup due to vectorization (CPU)
- CPU and GPU ...
  - ... achieved comparable performance
  - ... performance is sensitive to the underlying hardware
- GPU versions required more effort than their CPU counterparts
- Recommend OpenMP for the MEG application
- What might come...
  - ... increasing number of cores (e.g. NVIDIA's *Kepler*)
  - ... increasing SIMD support (e.g. Intel's *MIC*)
  - ... more powerful compilers (auto-vectorization/ -parallelization)
  - ... directive-based programming paradigms for GPUs

Thank you for your attention.

C - Gradient Routine

Kernel 1.1 cSparse

Input: x, λ, p, $f(x)$
Output: $f(x)$, z

1: **for** $j = 0 \rightarrow k - 1$ **do**
2: $tmp \leftarrow \lambda \cdot x_j$;
3: $f(x) \leftarrow f(x) + |tmp|^p$
4: // $\mathcal{O}(1)$ computations
5: $z_j \leftarrow tmp$
6: **end for**

Kernel 1.2 cDense

Input: L, x, b, p, $f(x)$
Output: $f(x)$, z

1: **for** $i = 0 \rightarrow n - 1$ **do**
2: $tmp \leftarrow \sum_{j=0}^{k-1} L_{i,j} \cdot x_j$;
3:
4: $tmp \leftarrow tmp - b_i$
5: $f(x) \leftarrow f(x) + |tmp|^p$
6: // $\mathcal{O}(1)$ computations
7:
8: **for** $j = 0 \rightarrow k - 1$ **do**
9: $z_j \leftarrow z_j + tmp \cdot L_{i,j}$
10: **end for**
11: **end for**

GradOuter Version

Kernel 1.3 gradOuter

Input: L, x, b, p, $f(x)$
Output: $f(x)$, z

1: **# omp for reduction(+:$f(x)$)**
2: **for** $i = 0 \rightarrow n - 1$ **do**
3: $tmp \leftarrow \sum_{j=0}^{k-1} L_{i,j} \cdot x_j$;
4:
5: $tmp \leftarrow tmp - b_i$
6: $f(x) \leftarrow f(x) + |tmp|^p$
7: // $\mathcal{O}(1)$ computations
8:
9: **for** $j = 0 \rightarrow k - 1$ **do**
10: $zLocal_j \leftarrow zLocal_j + tmp \cdot L_{i,j}$
11: **end for**
12:
13: // Reduce *zLocal* to z
14: **end for**

GradInner Version

Kernel 1.4 gradHead

Input: $L,\ x,\ h,\ b,\ p,\ f(x)$
Output: $f(x),\ h$

1: # **omp for reduction**$(+{:}f(x))$
2: **for** $i = 0 \rightarrow n-1$ **do**
3: $\quad tmp \leftarrow \sum_{j=0}^{k-1} L_{i,j} \cdot x_j$;
4:
5: $\quad tmp \leftarrow tmp - b_i$
6: $\quad f(x) \leftarrow f(x) + |tmp|^p$
7: $\quad$ // $\mathcal{O}(1)$ computations
8: $\quad h_i \leftarrow tmp$
9: **end for**

Kernel 1.5 gradTail

Input: L, h, z
Output: z

1: **for** $i = 0 \rightarrow n-1$ **do**
2: $\quad$ # **omp for**
3: $\quad$ **for** $j = 0 \rightarrow k-1$ **do**
4: $\quad\quad z_j \leftarrow z_j + h_i \cdot L_{i,j}$
5: $\quad$ **end for**
6: **end for**

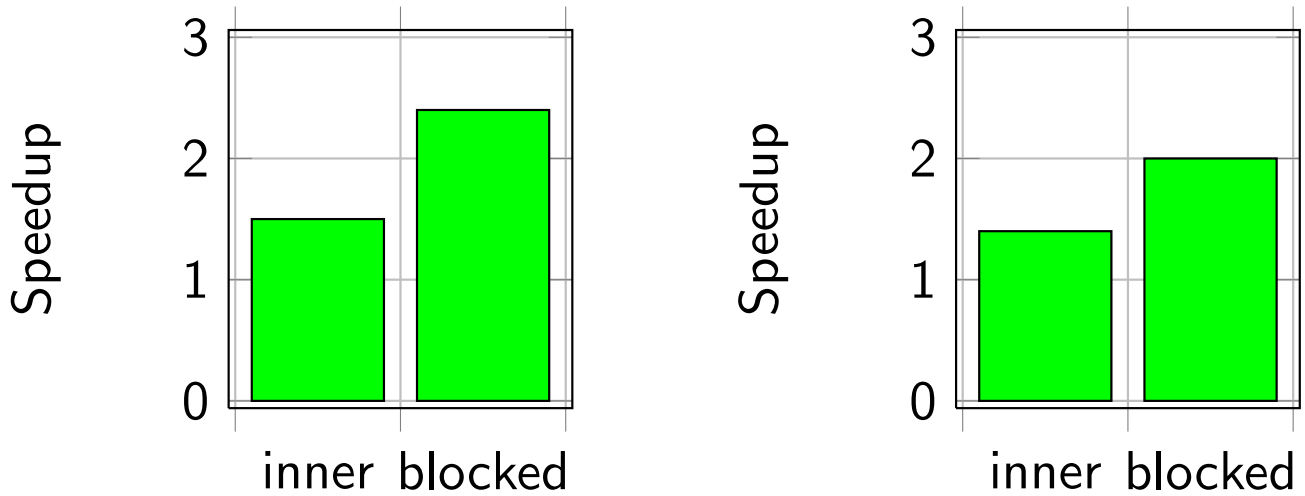
Gradient Routine

Kernel 1.6 gradTail

Input: L, h, z
Output: z

1: $numBlocks \leftarrow k/BLOCKDIM$
2: **#pragma omp for**
3: **for** $i = 0 \rightarrow numBlocks - 1$ **do**
4: /* Initialize *tmp* with 0 */
5: **for** $j = 0 \rightarrow n - 1$ **do**
6: **for** $l = 0 \rightarrow BLOCKDIM - 1$ **do**
7: $tmp_l \leftarrow tmp_l + h_j \cdot L_{j,i \cdot BLOCKDIM+l}$;
8: **end for**
9: **end for**
10:
11: **for** $l = 0 \rightarrow BLOCKDIM - 1$ **do**
12: $z_{i \cdot BLOCKDIM+l} \leftarrow tmp_l$
13: **end for**
14: **end for**

Performance Results - CPU



(a) Single precision (b) Double precision

Figure: Speedup of the *inner* and *innerBlocked* versions over the *outer* version.

- Successive improvements

Gradient Routine

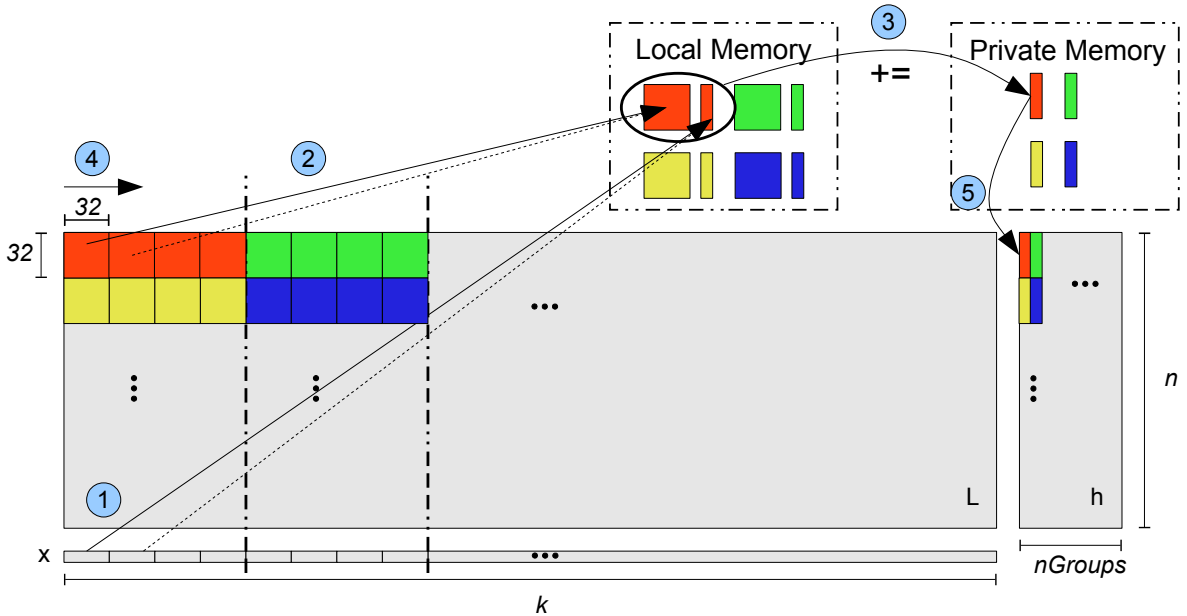


Figure: Kernel *gradHead*. Work-groups are colored differently.

- Reuses x
- Pays attention to coalescing
- Avoids bank conflicts

Gradient Routine

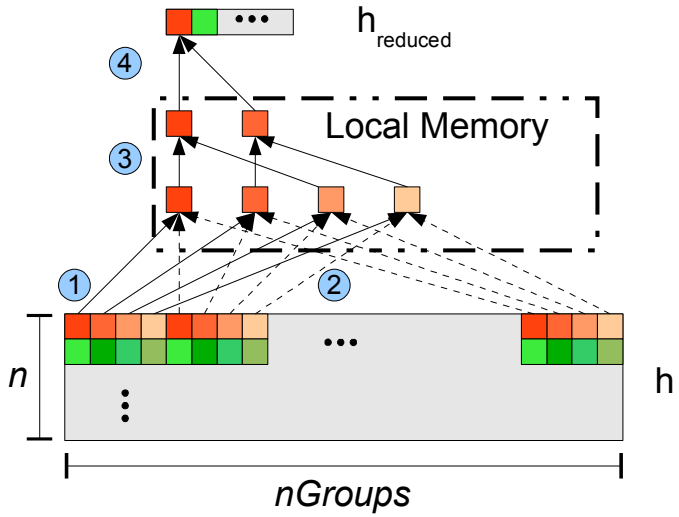


Figure: Kernel *gradReduce*. Work-items are colored differently.

- Pays attention to execution-path divergence
- Pays attention to coalescing

Gradient Routine

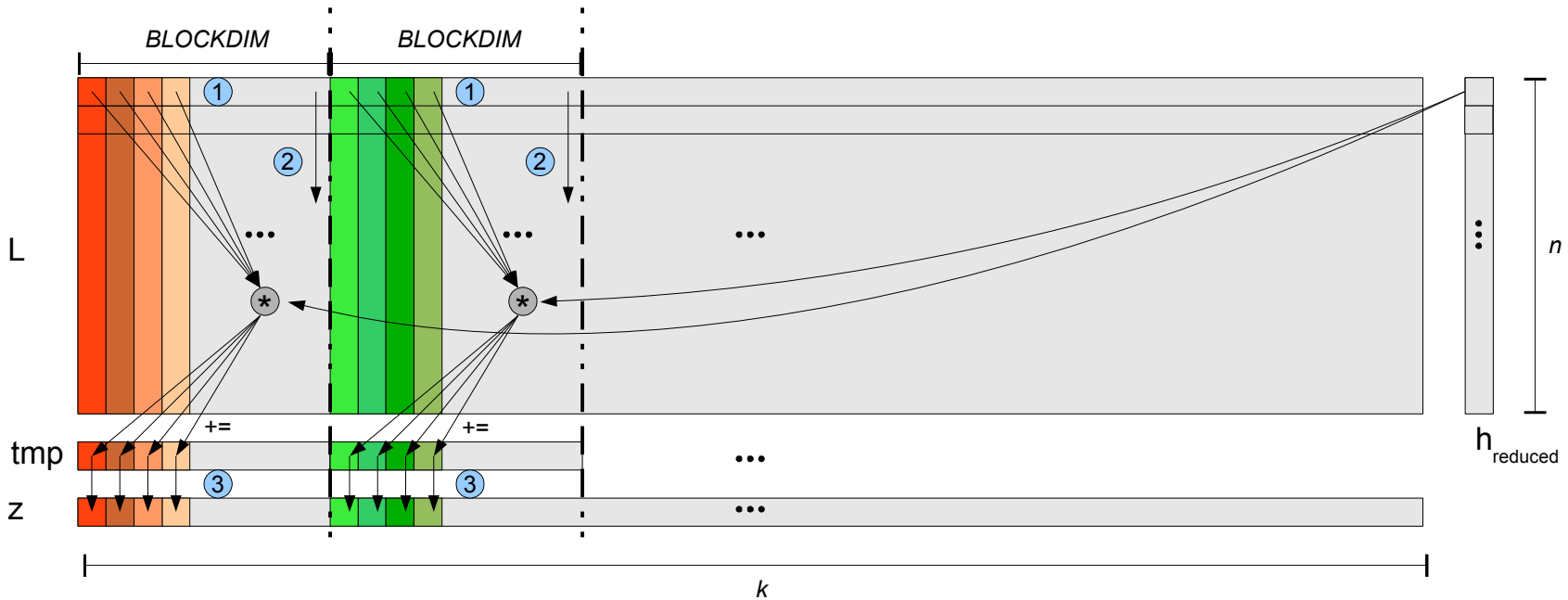
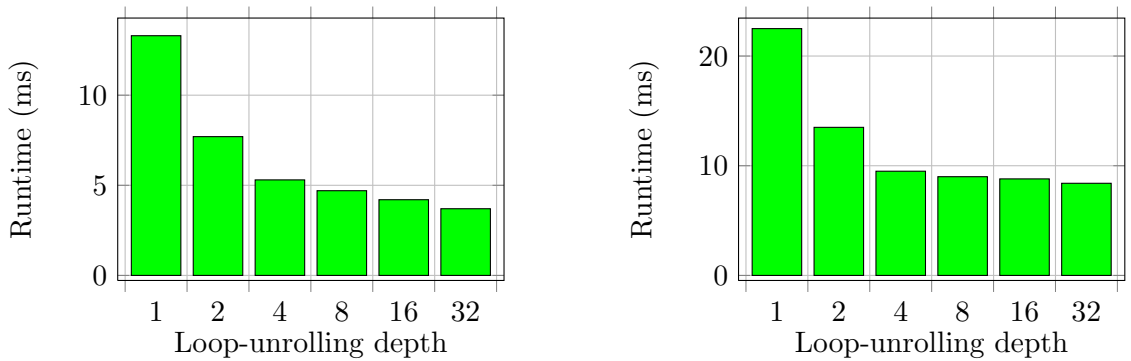


Figure: Kernel *gradTail*. Work-items are colored differently.

- Pays attention to coalescing
- High task-level parallelism (loop unrolled)

Performance Results - GPU



(a) Single precision (b) Double precision

Figure: Kernel *hesseHead* using CUDA. Impact of loop-unrolling on the runtime.

- Speedup due to loop-unrolling is significant
- Register pressure increases with increasing loop-unrolling depth
- OpenCL is more modest in terms of register usage than CUDA
- Register pressure did not limit thread-level parallelism
  - Limited by local-memory usage

References I


H.M. Bücker and R. Beucker.
Using Automatic Differentiation for the Solution of the Minimum p-Norm Estimation Problem in Magnetoencephalography.
Simulation Modelling Practice and Theory, 12(2):105–116, 2004.


H.M. Bücker, R. Beucker, and A. Rupp.
The NINA Software Package: Software for the Solution of Neuromagnetic Inverse Large-scale Problems, 2011.
Preliminary Manual.


M. Hämäläinen, R. Hari, R.J. Ilmoniemi, J. Knuutila, and O.V. Lounasmaa.
Magnetoencephalography—Theory, Instrumentation, and Applications to Noninvasive Studies of the Working Human Brain.
Reviews of modern Physics, 65(2):413, 1993.

References II


Intel.
A Guide to Auto-Vectorization with Intel C++ Compilers.
http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/, October 2011.


Intel.
Writing Optimal OpenCL Code with Intel OpenCL SDK.
http://software.intel.com/file/39189, February 2012.


NVIDIA.
NVIDIA's Next Generation CUDA Compute Architecture: Fermi.
NVIDIA Whitepaper, 2009.


NVIDIA.
CUDA C Best Practices Guide, March 2012.
Version 4.0.