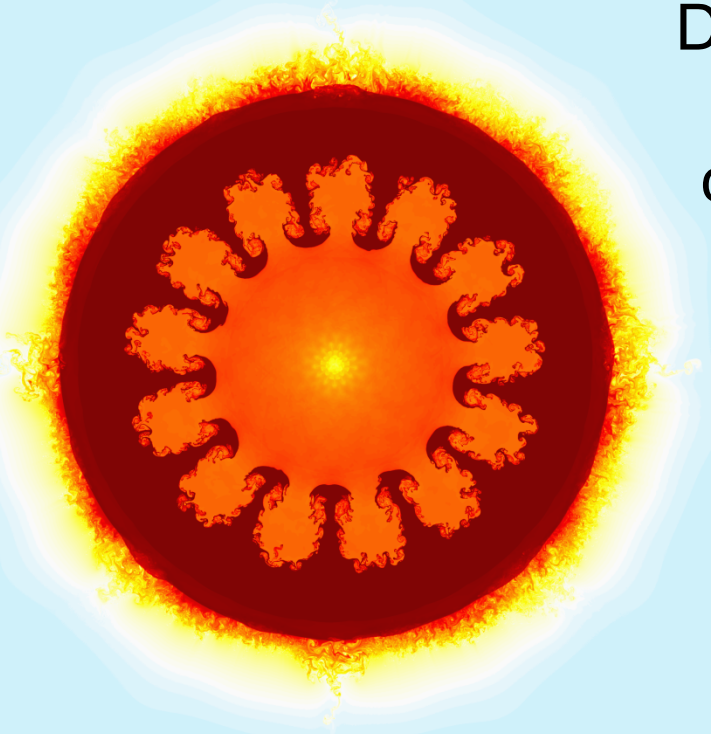
A Study of the Performance of Multifluid PPM Gas Dynamics on CPUs and GPUs

Pei-Hung Lin, Jagan Jayaraj, Paul R. Woodward
LCSE, University of Minnesota

UNIVERSITY OF MINNESOTA

- Began by working with Cell processor.
- Put entire PPM gas dynamics code on Cell
- Scaled linearly to 8000 Cell processors on Roadrunner at Los Alamos.
- Delivered performance of whole application on Intel or IBM CPU = **18%-16%** of 32-bit peak performance (26.0– 38.3 Gflop/s).
- Can we adapt this methodology to the GPU?

We are working with a full, 3-D PPM gas dynamics code, which tracks multiple fluid constituents in the presence of strong shocks. Here we show this code, run on a workstation cluster (Intel Nehalem CPUs), simulating a test problem devised by David Youngs in 2007 to correspond to the unstable compression of fuel in a laser fusion capsule. We will study this CFD code to evaluate the potential benefits of the GPU.



We begin by taking only the best part of this very complicated algorithm – PPM advection. Then we study the entire algorithm.

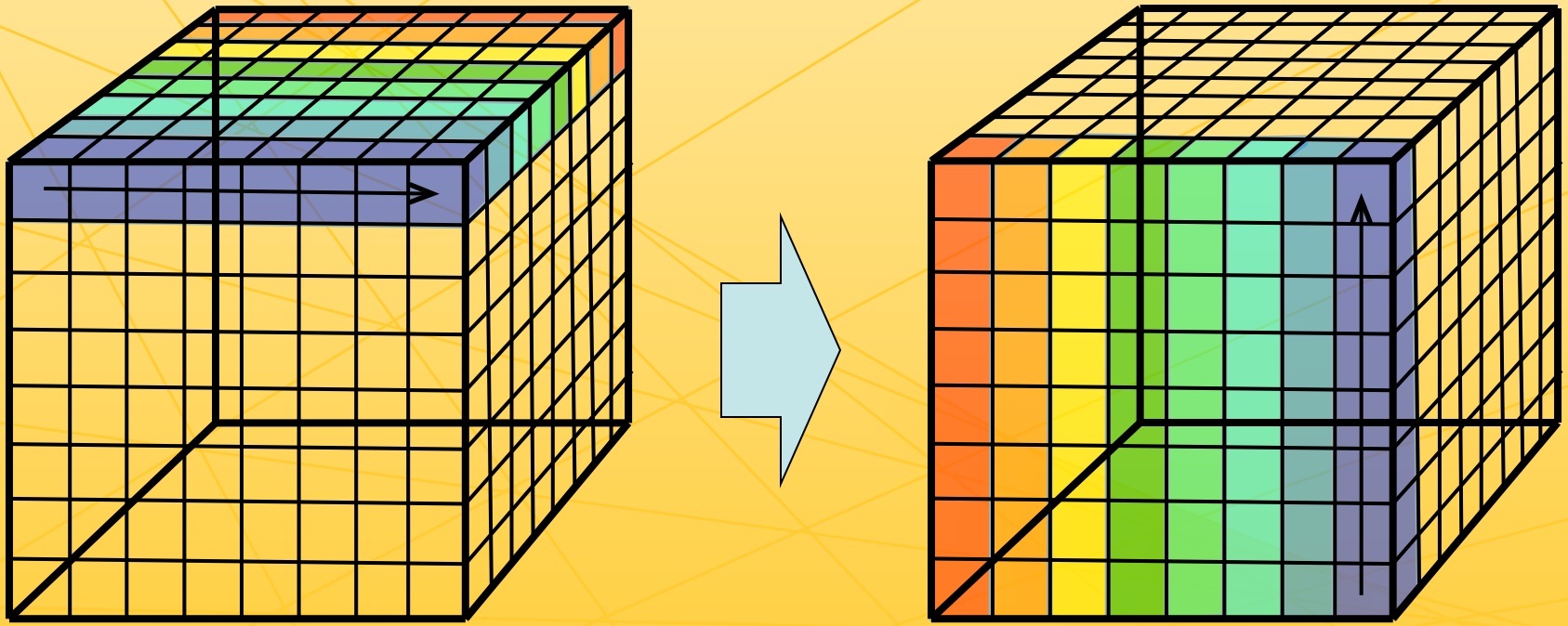
Programming Strategy

Key technique is organization of main memory data into briquettes of 4^3 cells.

This naturally produces operands that are both ***aligned*** and exact multiples of SIMD engine width:

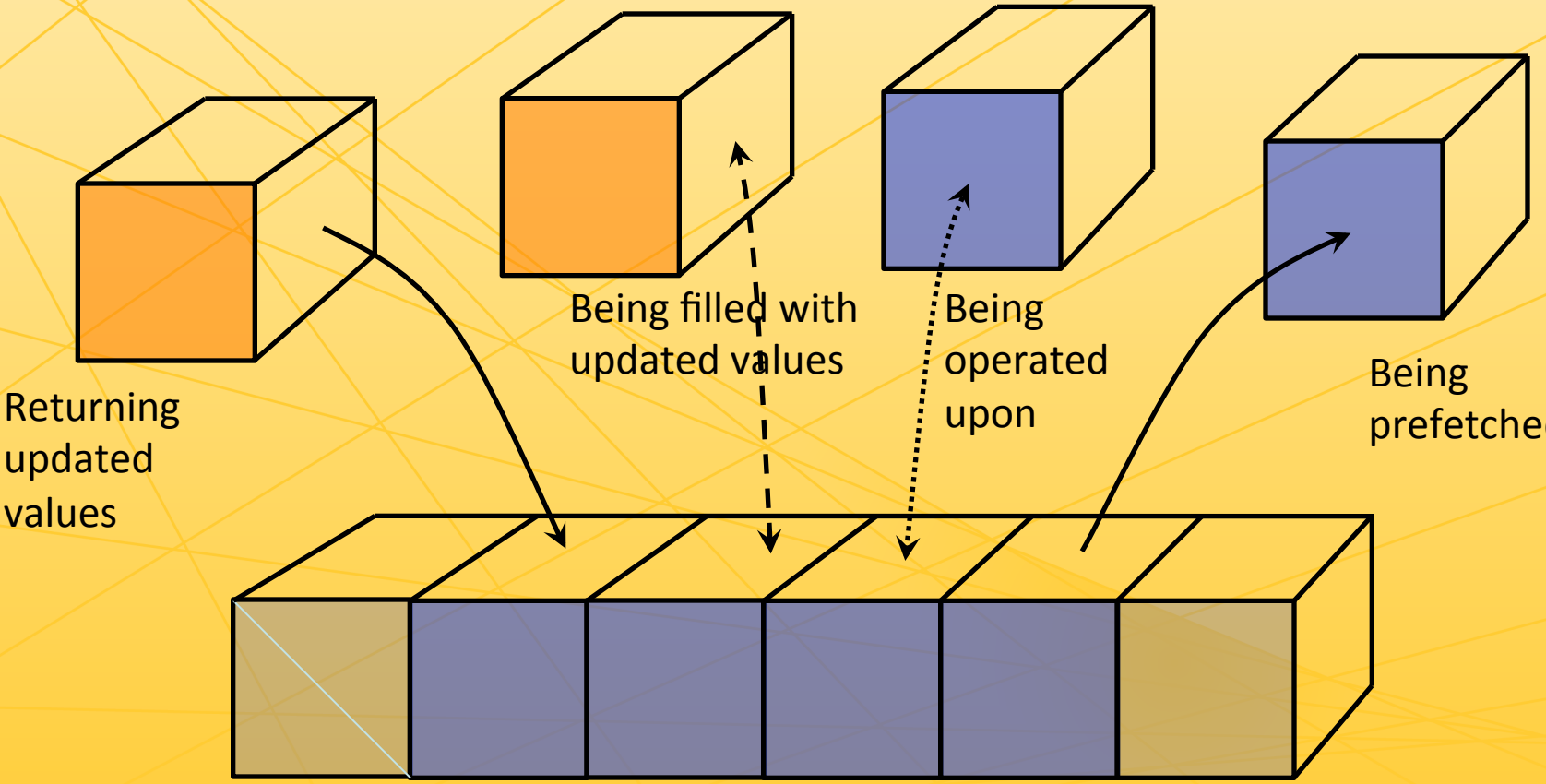
DDDbrick(**4,4,4,15**, nbqx, nbqy, nbqz,)

1. 1-D sweep computation.
2. Data is transposed for the following 1-D sweep.
3. X → Y → Z → Z → Y → X, having only 4 transpositions in every 6 passes.



We prefetch one data record while we unpack and operate on the previous one cached in our local store.

We write back one updated data record while we fill in values in another one cached in our local store.



The briquette records shown at the bottom are in main memory, while those above are in the local store.

PPM Advection

- Small application code extracted and based on PPM gas dynamic algorithm.
- Best-case subroutine in code for GPU.
- Read 1 word, do 89 flops, write 1 word.
- Do brick of 448^3 cells.
- **37.8** Gflop/s on dual quad-core Intel Nehalem processors, 20.1% of the 32-bit peak performance

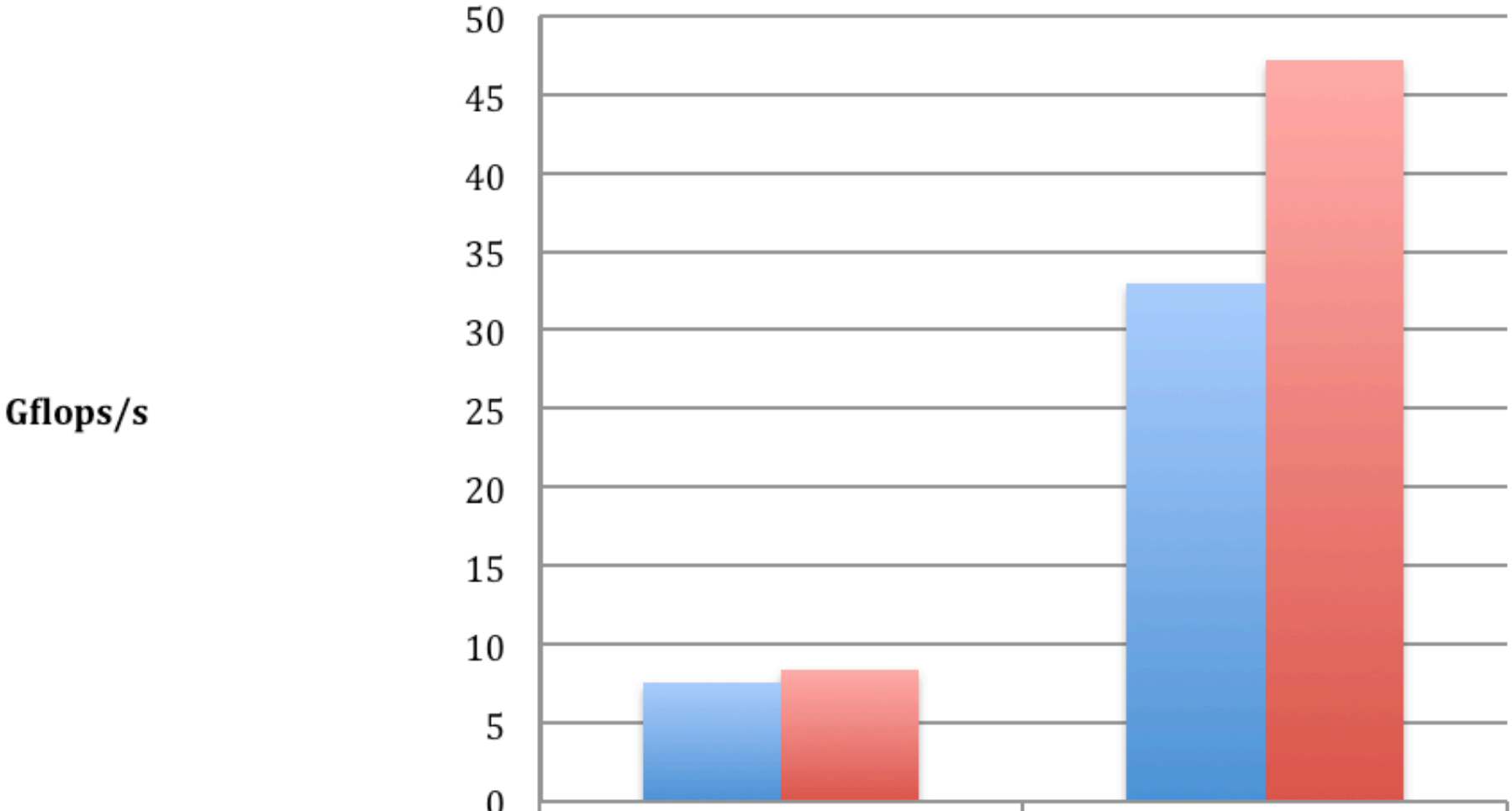
GPU Implementation

- Each thread block requires 13.25 KB of shared memory, permitting only 3 thread blocks per 2 SIMD engines (1 SM).
- First attempt gave only 7.25 Gflops/s (Fermi).
- Recomputation of temporaries got shared mem. to 5.56 KB, permitting 4 thread blocks per SIMD eng.
- cudaMemcpy slow for our briquettes, so built own kernels for boundary communication.

Advection Performance on GPU

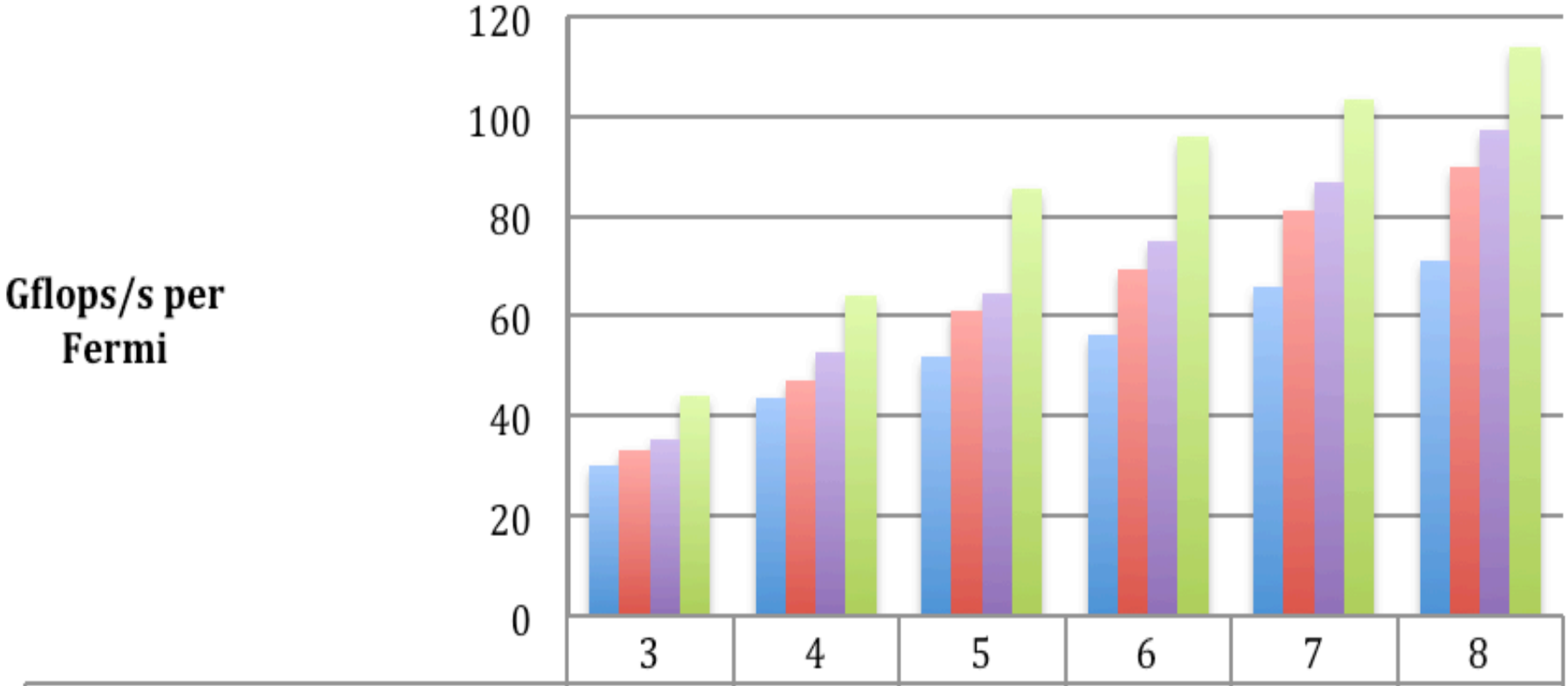
- Delivered performance = **89.8 Gflop/s** for the Fermi GPU's 28 SIMD engines, 8.7% of the 32-bit peak.
- With –fast-math compiler option, the same code delivers 114 Gflops/s. We cannot take advantage of this 27% performance increase, because of the accumulation of error over time steps.
- With 28 SIMD engines, each 16 wide, Fermi GPU outruns two Intel quadcore CPUs by a factor of **2.6**. (28/8 = 3.5)
- The delivered performance increases nearly linearly with the number of thread blocks per SIMD engine, to a maximum of 3.21 Gflop/s/SIMD-engine. Maximum determined by on-chip memory capacity.

cudaMemcpy V.S. data-copy kernel

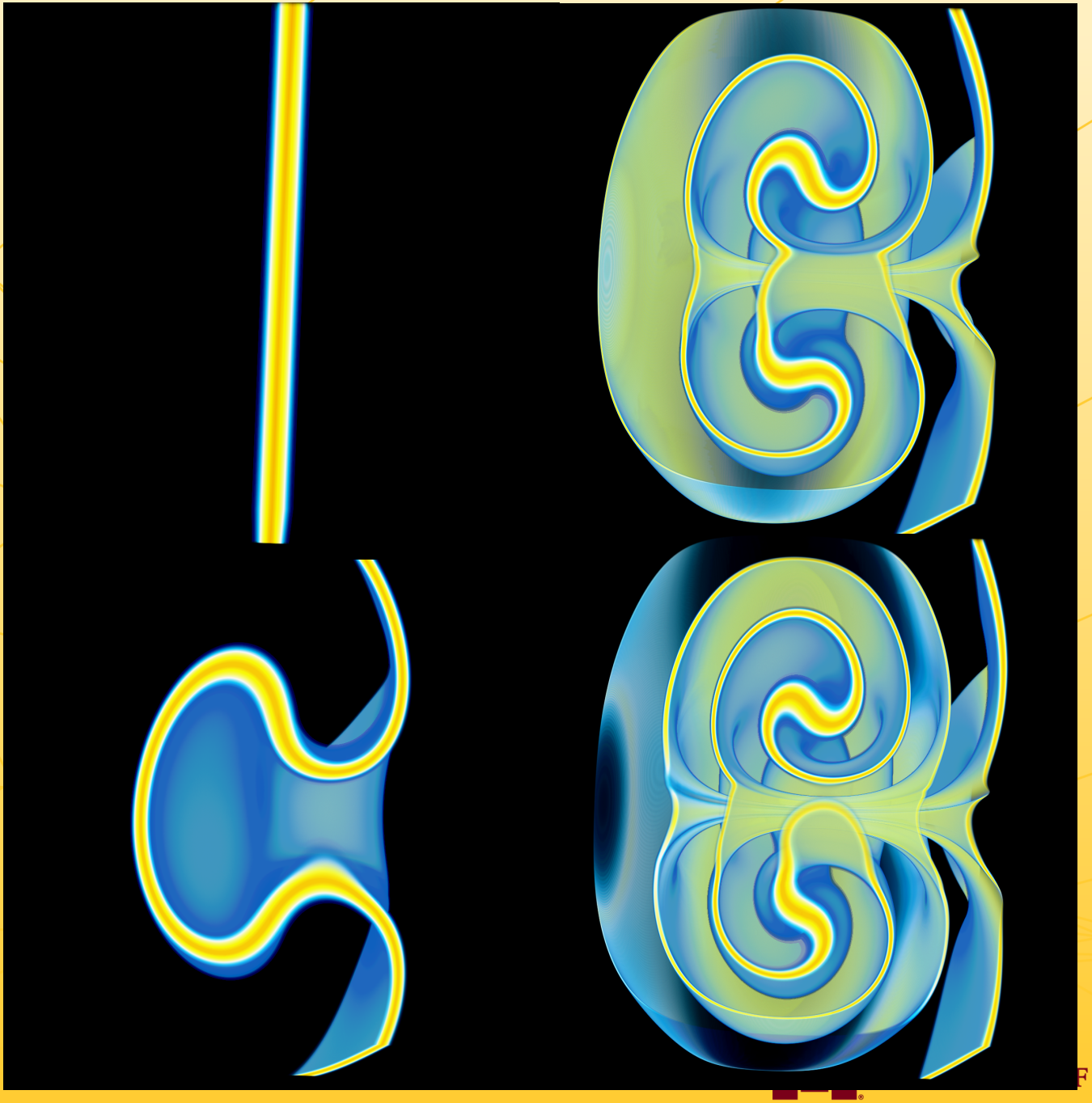


	cudaMemcpy	data-copy kernel
256^3, 3 thread blocks in SM	7.526	32.939
256^3, 4 thread blocks in SM	8.324	47.178

Advection example performance



	3	4	5	6	7	8
256^3, constdt = .00052601	30.107	43.448	51.827	56.247	65.962	71.135
256^3, constdt = .000052601	32.939	47.178	61.173	69.197	81.064	89.76
512^3, constdt = .000052601	35.143	52.763	64.741	75.049	86.947	97.416
256^3 constdt=.000052601-fast-math	43.985	64.137	85.642	95.784	103.256	113.743



Multifluid Gas Dynamics

- In the **Advection** example, each cell reads in 1 number, processes 60 flops, and writes back 1 number. (**30 flops/word**)
- For **PPM+PPB**, each cell has to read in 35 numbers, processes 1676 flops, and writes back 15 numbers. (**33.5 flops/word**)
- **The computation is too large** to be hosted on a single GPU chip. The multi-fluid gas dynamic application becomes memory-bandwidth bound for GPU.

Multifluid Gas Dynamics on CPU

- About 60 KB data workspace in cache, and 60 KB for instructions. Both fit into 256K L2.
- On Intel Westmere, single thread/core with vector length of 16 delivers 92% of the performance with 2 threads/core. Thread number per core isn't that important here.
- We measure the individual performance of each subroutine by running the single routine for many iterations while running all others only once.

Kernel	Flops/cell	Input	Output	intensity	Derived performance	Measured Performance
difuze1	20	14	2	1.25	**25.7**	25.8
difuze2	203	25	2	7.52	**45.2**	42.8
fvscses	84	7	23	2.8	**62.5**	56.6
interpriemann	25	10	10	1.25	**42.8**	40.8
ppmdntrf0vec	32	3	3	5.33	**45.4**	43.0
ppmdntrfavec	68	5	3	8.5	**38.8**	38.0
ppmintrf0vec	34	3	5	4.25	**39.0**	37.6
interprhopux	160	18	16	4.71	**44.0**	41.2
riemannstates	170	13	8	8.29	**29.3**	29.2
Riemann0+1	110	18	2	5.5	**94.9**	79.9
ppb10constrainx	78	3	3	13	**24.4**	24.5
ppb10fv	105	4	18	4.77	**67.8**	60.6
fluxes	106	31	13	2.41	**73.1**	64.5
Cellupdate	126	28	23	2.47	**79.6**	70.4
difussion	124	25	23	2.58	**74.2**	65.2

Performance on CPU

- Delivered performance is **25.98** Gflops/s for single Westmere CPU.
- If we eliminate the cost of fetching/writing data, the performance rises to **37.32**
- The derived performances are “never to be exceeded” for each single routine. If we take average of those numbers, we get **44.4** Glops/s for single Westmere CPU.

Multifluid Gas Dynamics on GPU

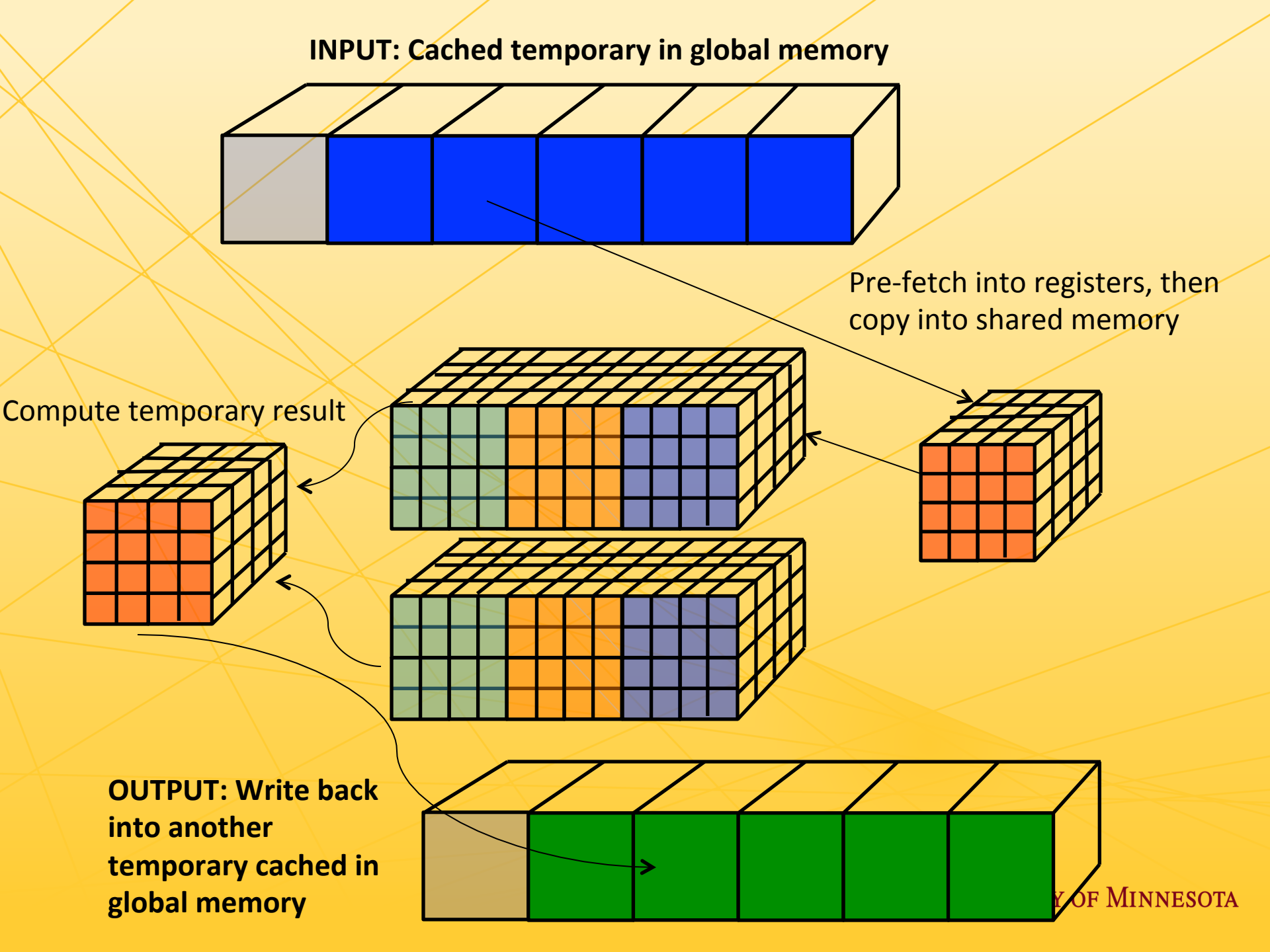
- GPU implementation is started from a highly optimized program expression on CPU.
- Vector operands are aligned and with vector length of 64.
- Data workspace is reduced, unnecessary data traffic is eliminated.
- 64 GPU threads, in a single thread block, are assigned to update a strip of 64 briquettes of 4x4x4 cells each.

Kernel Implementation

- Kernel launching & global barrier are costly, our entire code (except for boundary communication copies) is implemented as a single Nvidia kernel.
- We decompose the code into sub-kernels and each has its explicit input and output.
- Sub-kernels are subroutines called from the main Nvidia kernel. NVCC inlines all of them in the compilation.
- Each kernel is designed to have as few input/ output as possible. Iterations of optimization are applied to reduce the input and output for each sub-kernel.

The Need to Spill Temporary Data

- The outputs from one sub-kernel are stored in global memory and serve as input for later sub-kernels.
- L1 cache + shared memory is too small to keep all of them. We are forced to spill the results into the global memory.
- Each sub-kernel executes on a grid pencil of 64 grid briquettes arranged in a line.
- Much of the shared memory is reserved for pre-fetching inputs and scratch arrays in sub-kernels. Due to the 48 KB limitation, a maximum of only **15** inputs are prefetched.


INPUT: Cached temporary in global memory
Pre-fetch into registers, then copy into shared memory
Compute temporary result
OUTPUT: Write back into another temporary cached in global memory

Variants of PPM advection Kernels

Kernel	Flops/cell	Input	Output	intensity	Performance	Global memory throughput
interpriemann	25	10	10	1.25	**7.16**	24.42
ppmdntrf0vec	32	3	3	5.33	**28.88**	18.50
ppmdntrf0vec	32	3	3	5.33	**28.88**	18.50
ppmdntrfavec	68	5	3	8.5	**27.25**	12.49
ppmdntrfavec	68	5	3	8.5	**27.25**	12.49
ppmintrf0vec	34	3	5	4.25	**29.93**	24.10
ppmintrf0vec	34	3	5	4.25	**29.93**	24.10
interprhopux	160	18	16	4.71	**18.69**	14.44

The performance numbers are roughly a third of the levels reported for the advection code.

Coupled equations

- Computational intensities are lower than for the advection code by factors of 3 or 4. This results from their use within a code that is solving *coupled* partial differential equations.
- Each PPM interpolation kernel produces as output the 3 coefficients of its interpolation parabola for use later in the code.
- There is cross coupling between the interpolations themselves.
- The first kernel ingests the fundamental fluid state variables together with sound speed and other values.
- In the final kernel we put these quantities back together to produce consistent, constrained interpolation parabola for the densities, pressure, and velocity components.

Kernel	Flops/cell	Input	Output	intensity	Performance	Global memory throughput
Difuze1+2	223	25	2	8.26	42.41	21.25
fvscses1	36	6	11	2.12	17.74	32.31
fvscses2	48	7	15	2.18	20.61	33.50
riemannstates0	54	11	5	3.38	20.43	23.19
riemannstates1	66	13	8	3.14	10.65	12.84
Riemann0	9	8	2	0.90	11.96	51.35
riemann1	101	18	2	5.05	42.30	18.67
ppb10constrainx	78	3	3	13.00	43.92	12.97
ppb10fv	105	4	18	4.77	26.59	22.01
ppb10fvy	88	20	4	3.67	34.77	38.19
ppb10fvz	58	20	3	2.52	23.46	37.41
fluxesrhop	78	20	7	2.89	11.38	15.19
fluxesuten	28	16	6	1.27	9.14	28.39
Cellupdate	126	28	28	2.47	10.90	16.78
difussion	124	25	25	2.58	7.50	11.35

Performance on GPU

- As our first attempt to implement such large computation on GPU, a overall performance is 14.1 Gflops/s.
- Excluding two sub-kernels without pre-fetching, the performance is about 14.93 Gflops/s.
- The time-weighted average performance of all sub-kernels is 22.45 Gflops/s.

Reasons for the slow speed

- Temporaries are forced to spill into global memory, and there is tremendous traffic to and from the global memory.
- Huge register spilling reported by compiler. However, some of this spilling could be unnecessary and caused by inappropriate compiler heuristics.
- We ask for only 4 thread blocks in each SM to get 12 KB shared memory in each thread block.
- For the advection code, this # of thread blocks delivered only half our best performance.
- With more one-chip data storage capacity, we might be able to run much faster.

What would a fast GPU implementation take?

- We could return to earlier versions of PPM, which do not so tightly couple the differential equations.
  Trades simulation quality for GPU speed.
- We could very substantially reorganize the computational flow in a single grid briquette update to reduce spilling to global memory.
- We could somehow incorporate the boundary communications into our present kernel to eliminate possibly unnecessary global barriers.
- If we could invalidate texture cache entries, we could exploit the texture memory on the chip.

What GPU changes would make our application run better?

- More on-chip data storage per SIMD unit.
- Modifications to make each SIMD engine run well with fewer, perhaps only a single, thread block (Westmere can do this).
  [This may require the previous change.]
- We do not need cache coherency, but we do need the texture cache to act as a normal cache in the sense that we can write to our own cache (no other SIMD engine will read this data).

Conclusion

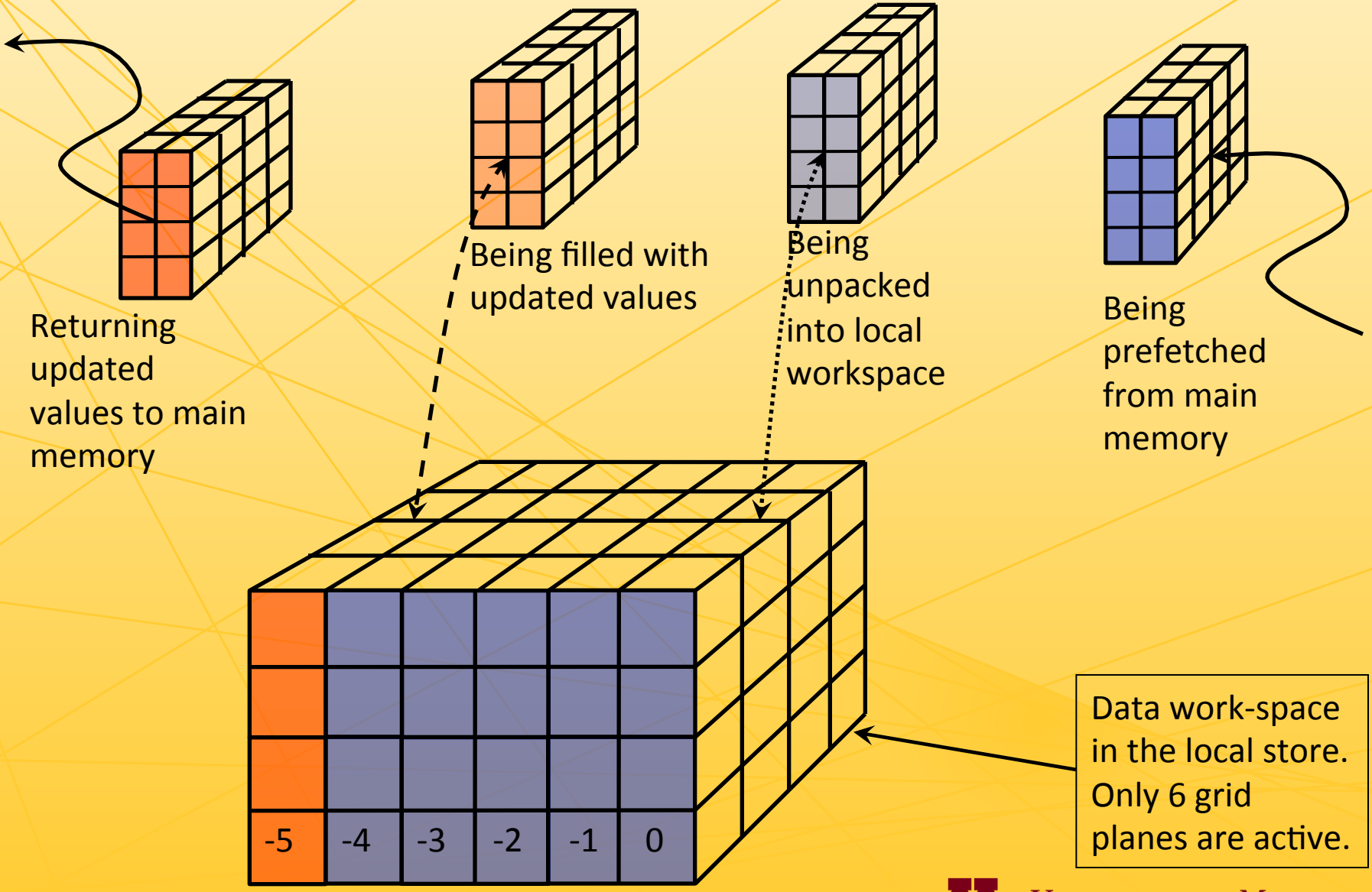
- The balance on the chip between data storage and computational resources is important.
- The lack of on-chip storage on a per-thread-block basis has forced us to spill intermediate results to GPU global memory. The result is data traffic that limits overall performance.
- Our application will run better with more on-chip memory per SIMD unit.

Questions and comments

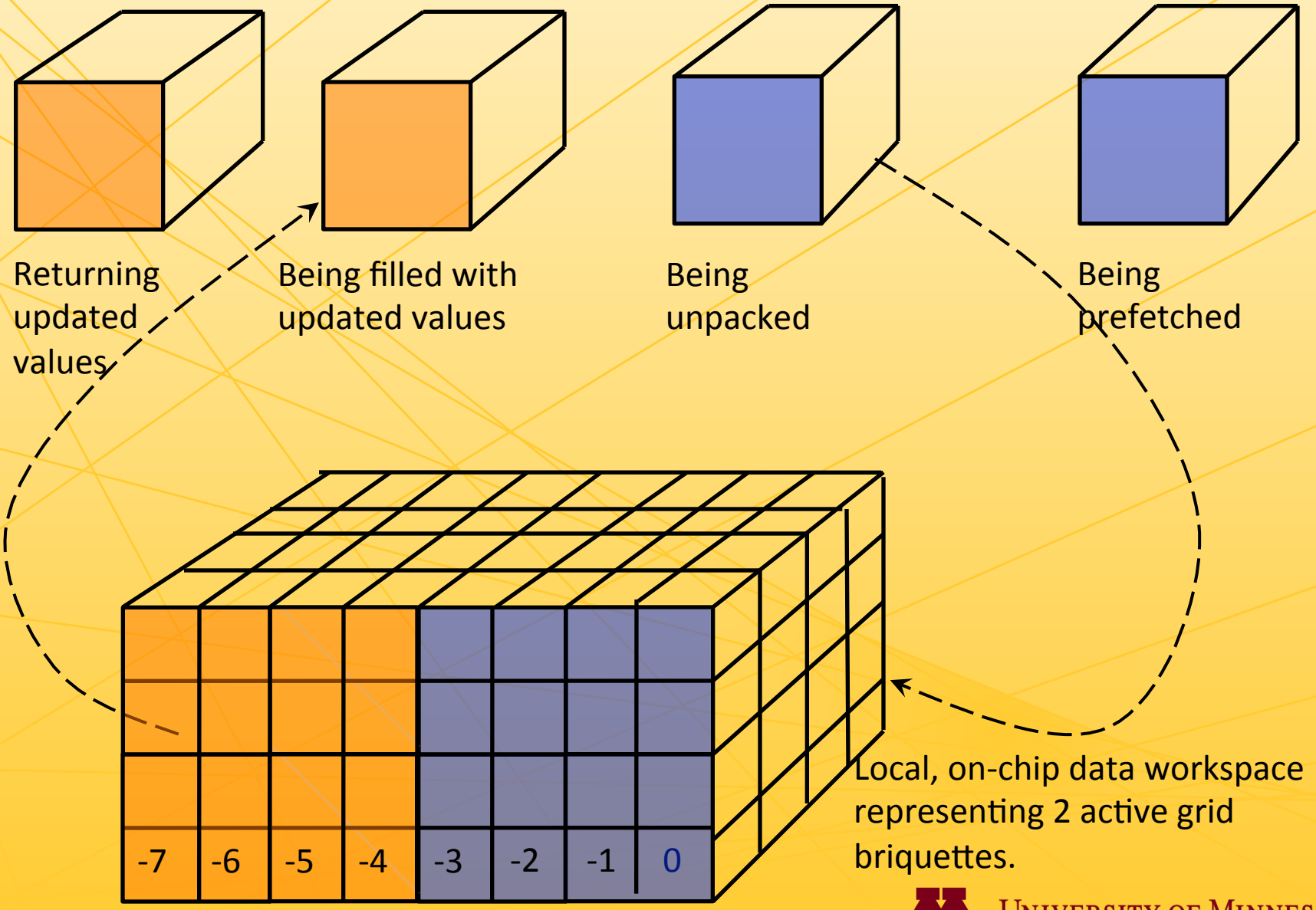
UNIVERSITY OF MINNESOTA

UNIVERSITY OF MINNESOTA

We build a grid-plane processing pipeline in the local store.



We build a briquette processing pipeline in the local store.



Future Work

- We have been advocating the briquette data structure, massive code pipelining, and short, aligned vector operands of the code restructuring and transformation.
- Automatic code translators to perform code restructuring transformations are under continued development.
- Minimizing cached inputs and outputs of code segments or kernels is a new demand that the GPU produces.
- We will attempt to extend our code translation tools to assist the programmer in addressing this new issue in code design.