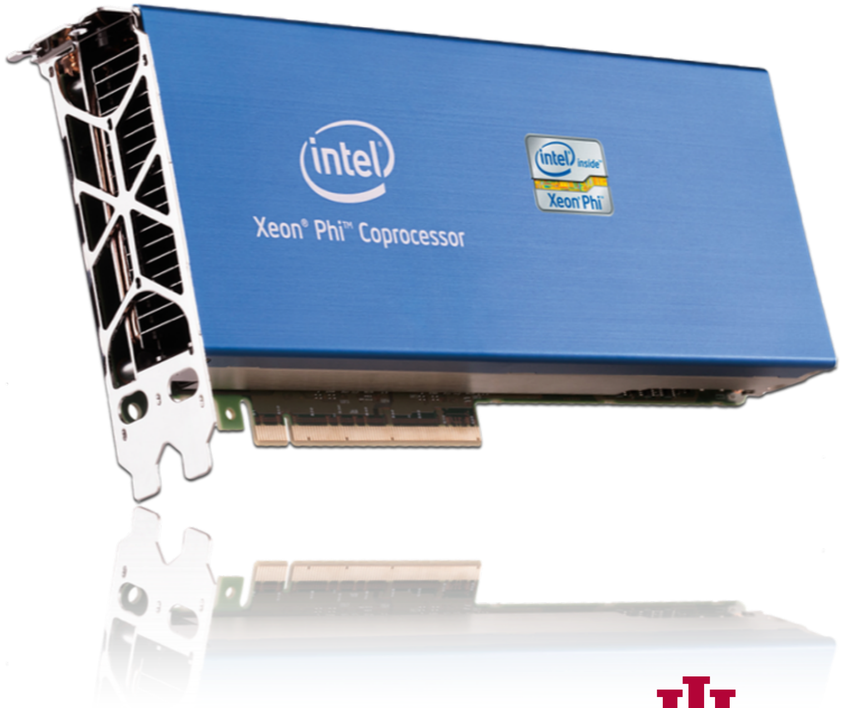
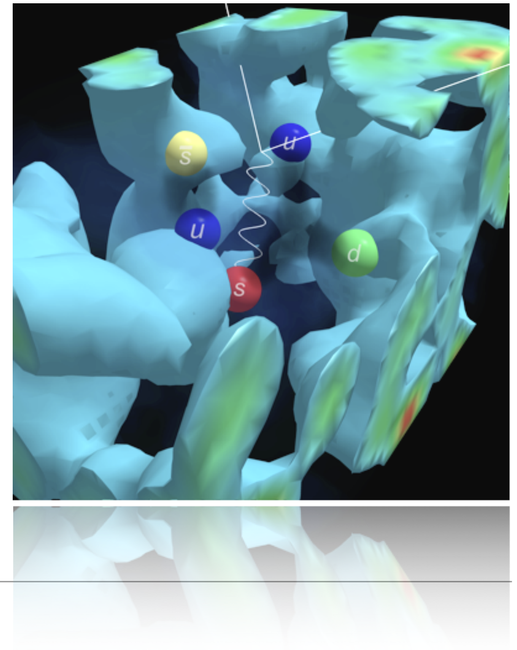
GPU vs Xeon Phi: Performance of Bandwidth Bound Applications with a Lattice QCD Case Study



Mathias Wagner

INDIANA UNIVERSITY

Lattice Quantum ChromoDynamics



and Deep Learning …

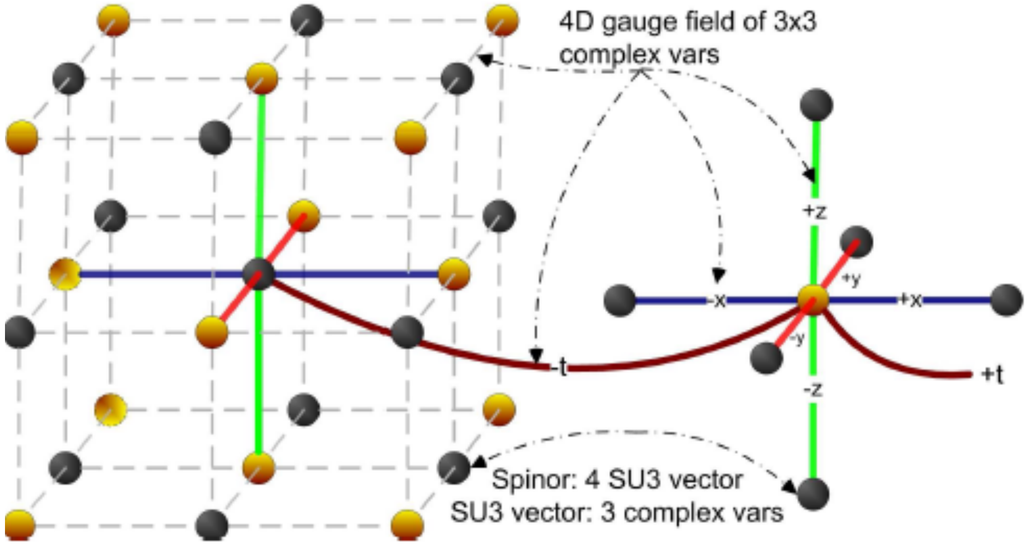
… sorry, not (yet?) here.

Lattice QCD: Some Basics

- QCD partition function
- 4 dimensional grid (=Lattice)
  - quarks live on lattice sites
  - gluons live on the links
- typical sizes: 24^3 x 6 to 256^4
  - parallelization over lattice sites (10^5 to 10^9)

$$Z_{\mathrm{QCD}}(T,\mu) = \int DAD\bar{\Psi}D\Psi e^{-S_E(T,\mu)}$$

includes integral over space and time



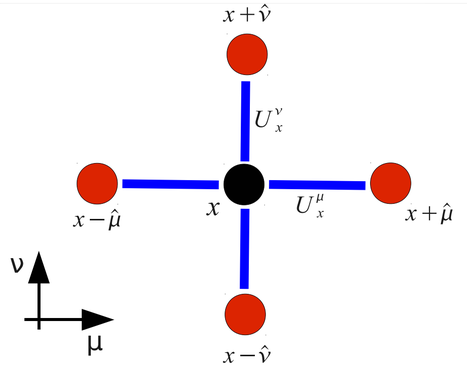
Staggered Fermion Matrix (Dslash)

- Krylov space inversion of fermion matrix dominates runtime
- within inversion application of sparse Matrix (Dslash) dominates (>80%)

Staggered Fermion Matrix (Dslash)

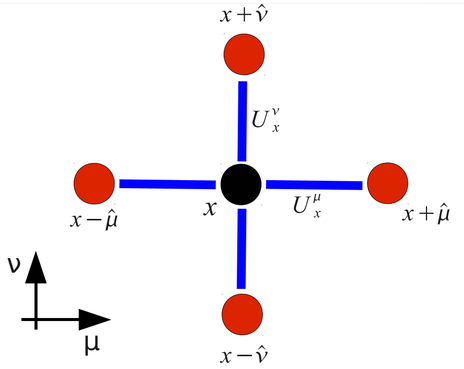
- Krylov space inversion of fermion matrix dominates runtime
- within inversion application of sparse Matrix (Dslash) dominates (>80%)
- Highly Improved Staggered Quarks (HISQ) use next and 3rd neighbor stencil

$$w_x = D_{x,x'} v_{x'} = \sum_{\mu=0}^{3} \left[\left\{ U_{x,\mu} v_{x+\hat{\mu}} - U^{\dagger}_{x-\hat{\mu},\mu} v_{x-\hat{\mu}} \right\} + \left\{ N_{x,\mu} v_{x+3\hat{\mu}} - N^{\dagger}_{x-3\hat{\mu},\mu} v_{x-3\hat{\mu}} \right\} \right]$$



Staggered Fermion Matrix (Dslash)

- Krylov space inversion of fermion matrix dominates runtime
- within inversion application of sparse Matrix (Dslash) dominates (>80%)
- Highly Improved Staggered Quarks (HISQ) use next and 3rd neighbor stencil



$$w_x = D_{x,x'} v_{x'} = \sum_{\mu=0}^{3} \left[\left\{ U_{x,\mu} v_{x+\hat{\mu}} - U^{\dagger}_{x-\hat{\mu},\mu} v_{x-\hat{\mu}} \right\} + \left\{ N_{x,\mu} v_{x+3\hat{\mu}} - N^{\dagger}_{x-3\hat{\mu},\mu} v_{x-3\hat{\mu}} \right\} \right]$$

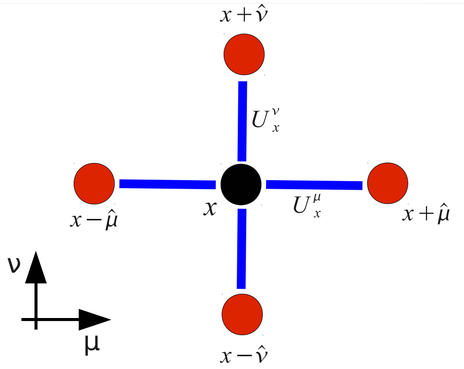
complex 3-dim vector
24 byte for fp32

complex 3x3 matrix
72 byte for fp32

complex 3x3 matrix + U(3) symmetry
56 byte for fp32

Staggered Fermion Matrix (Dslash)

- Krylov space inversion of fermion matrix dominates runtime
- within inversion application of sparse Matrix (Dslash) dominates (>80%)
- Highly Improved Staggered Quarks (HISQ) use next and 3rd neighbor stencil



$$w_x = D_{x,x'} v_{x'} = \sum_{\mu=0}^{3} \left[\left\{ U_{x,\mu} v_{x+\hat{\mu}} - U^{\dagger}_{x-\hat{\mu},\mu} v_{x-\hat{\mu}} \right\} + \left\{ N_{x,\mu} v_{x+3\hat{\mu}} - N^{\dagger}_{x-3\hat{\mu},\mu} v_{x-3\hat{\mu}} \right\} \right]$$

complex 3-dim vector
24 byte for fp32

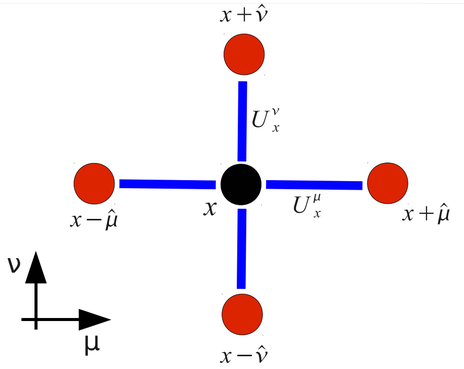
complex 3x3 matrix
72 byte for fp32

complex 3x3 matrix + U(3) symmetry
56 byte for fp32

- each site (x) loads 1024 bytes for links and 384 bytes for vectors, stores 24 bytes: total 1432 bytes / site
- performs 1146 flop: **arithmetic intensity: 0.8 flop/byte**

Staggered Fermion Matrix (Dslash)

- Krylov space inversion of fermion matrix dominates runtime
- within inversion application of sparse Matrix (Dslash) dominates (>80%)
- Highly Improved Staggered Quarks (HISQ) use next and 3rd neighbor stencil



$$w_x = D_{x,x'} v_{x'} = \sum_{\mu=0}^{3} \left[\left\{ U_{x,\mu} v_{x+\hat{\mu}} - U^{\dagger}_{x-\hat{\mu},\mu} v_{x-\hat{\mu}} \right\} + \left\{ N_{x,\mu} v_{x+3\hat{\mu}} - N^{\dagger}_{x-3\hat{\mu},\mu} v_{x-3\hat{\mu}} \right\} \right]$$

complex 3-dim vector
24 byte for fp32

complex 3x3 matrix
72 byte for fp32

complex 3x3 matrix + U(3) symmetry
56 byte for fp32

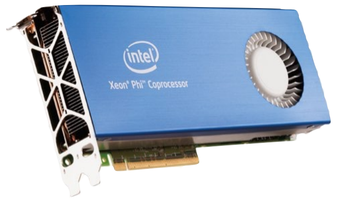
- each site (x) loads 1024 bytes for links and 384 bytes for vectors, stores 24 bytes: total 1432 bytes / site
- performs 1146 flop: **arithmetic intensity: 0.8 flop/byte**

sensitive to memory bandwidth

Accelerators

Sorry, not the ones with liquid helium cooling and TDP > 300W.

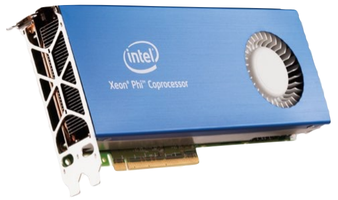
Intel Xeon Phi and Nvidia Tesla



	5110	7120	K20	K20X	K40
Cores / SMX	60	61	13	14	15
Vector instructions	512 bit (16 fp32)				
CUDA cores / SMX			192		
Clock Speed [MHz]	1053	1238 - 1333	705	732	745-875
peak fp32 [TFlop/s]	2.02	2.42	3.52	3.91	4.29
peak fp64 [TFlop/s]	1.01	1.21	1.27	1.31	1.43
Memory [GB]	8	8	5	6	12
Memory Bandwidth [GB/s]	320	352	208	250	288
L1 Cache [kB] / (Core/SMX)	32		16-48 + 48 (Texture)		
L2 Cache [MB]	30 (60 x 0.5)	30.5 (61 x 0.5)	1.5		
TDP [W]	225	300	225	235	235

Intel Xeon Phi and Nvidia Tesla

How can we achieve this performance?

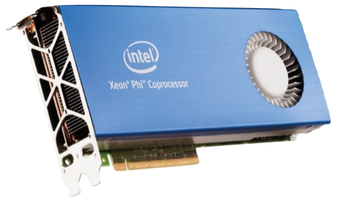


	5110	7120	K20	K20X	K40
Cores / SMX	60	61	13	14	15
Vector instructions	512 bit (16 fp32)				
CUDA cores / SMX			192		
Clock Speed [MHz]	1053	1238 - 1333	705	732	745-875
peak fp32 [TFlop/s]	2.02	2.42	3.52	3.91	4.29
peak fp64 [TFlop/s]	1.01	1.21	1.27	1.31	1.43
Memory [GB]	8	8	5	6	12
Memory Bandwidth [GB/s]	320	352	208	250	288
L1 Cache [kB] / (Core/SMX)	32		16-48 + 48 (Texture)		
L2 Cache [MB]	30 (60 x 0.5)	30.5 (61 x 0.5)	1.5		
TDP [W]	225	300	225	235	235

Intel Xeon Phi and Nvidia Tesla

How can we achieve this performance?

How can we saturate the available bandwidth?



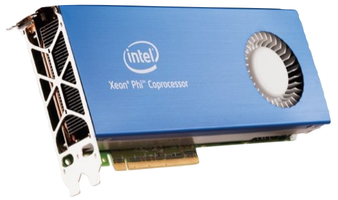
	5110	7120	K20	K20X	K40
Cores / SMX	60	61	13	14	15
Vector instructions	512 bit (16 fp32)				
CUDA cores / SMX			192		
Clock Speed [MHz]	1053	1238 - 1333	705	732	745-875
peak fp32 [TFlop/s]	2.02	2.42	3.52	3.91	4.29
peak fp64 [TFlop/s]	1.01	1.21	1.27	1.31	1.43
Memory [GB]	8	8	5	6	12
Memory Bandwidth [GB/s]	320	352	208	250	288
L1 Cache [kB] / (Core/SMX)	32		16-48 + 48 (Texture)		
L2 Cache [MB]	30 (60 x 0.5)	30.5 (61 x 0.5)	1.5		
TDP [W]	225	300	225	235	235

Intel Xeon Phi and Nvidia Tesla

How can we achieve this performance?

How can we saturate the available bandwidth?

How much energy does that require?



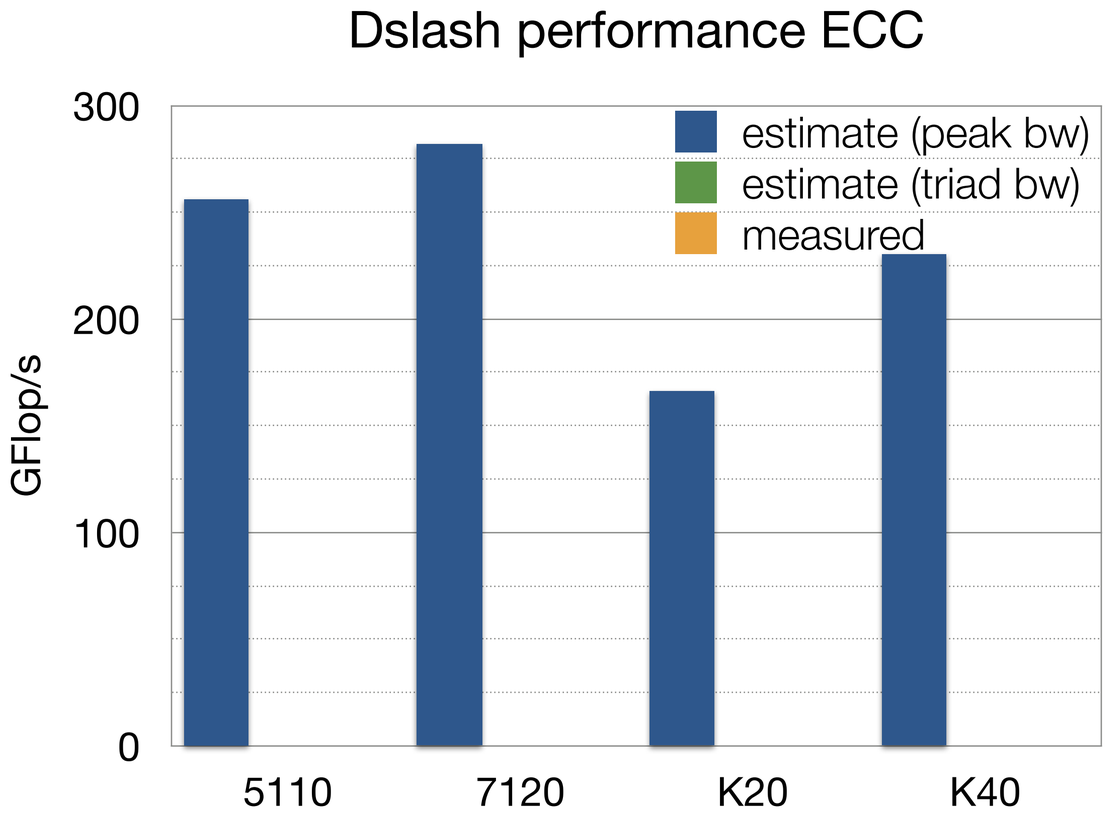
	5110	7120	K20	K20X	K40
Cores / SMX	60	61	13	14	15
Vector instructions	512 bit (16 fp32)				
CUDA cores / SMX			192		
Clock Speed [MHz]	1053	1238 - 1333	705	732	745-875
peak fp32 [TFlop/s]	2.02	2.42	3.52	3.91	4.29
peak fp64 [TFlop/s]	1.01	1.21	1.27	1.31	1.43
Memory [GB]	8	8	5	6	12
Memory Bandwidth [GB/s]	320	352	208	250	288
L1 Cache [kB] / (Core/SMX)	32		16-48 + 48 (Texture)		
L2 Cache [MB]	30 (60 x 0.5)	30.5 (61 x 0.5)	1.5		
TDP [W]	225	300	225	235	235

Setting the bar

What performance can we expect on the different accelerators?
Is our code optimized?

Estimated Dslash Performance

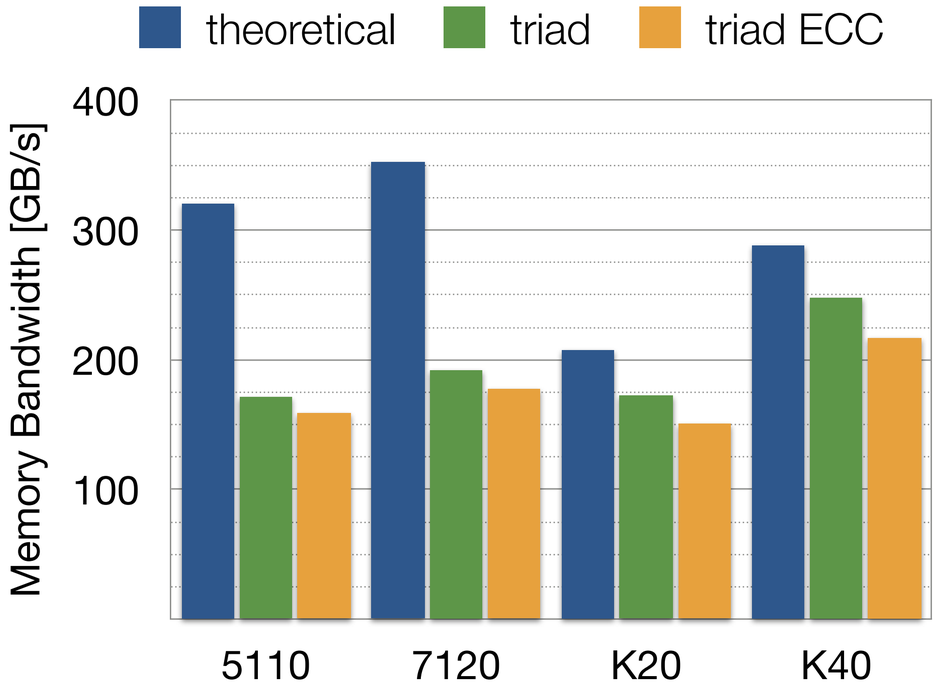
- naive model:
  bandwidth times arithmetic intensity

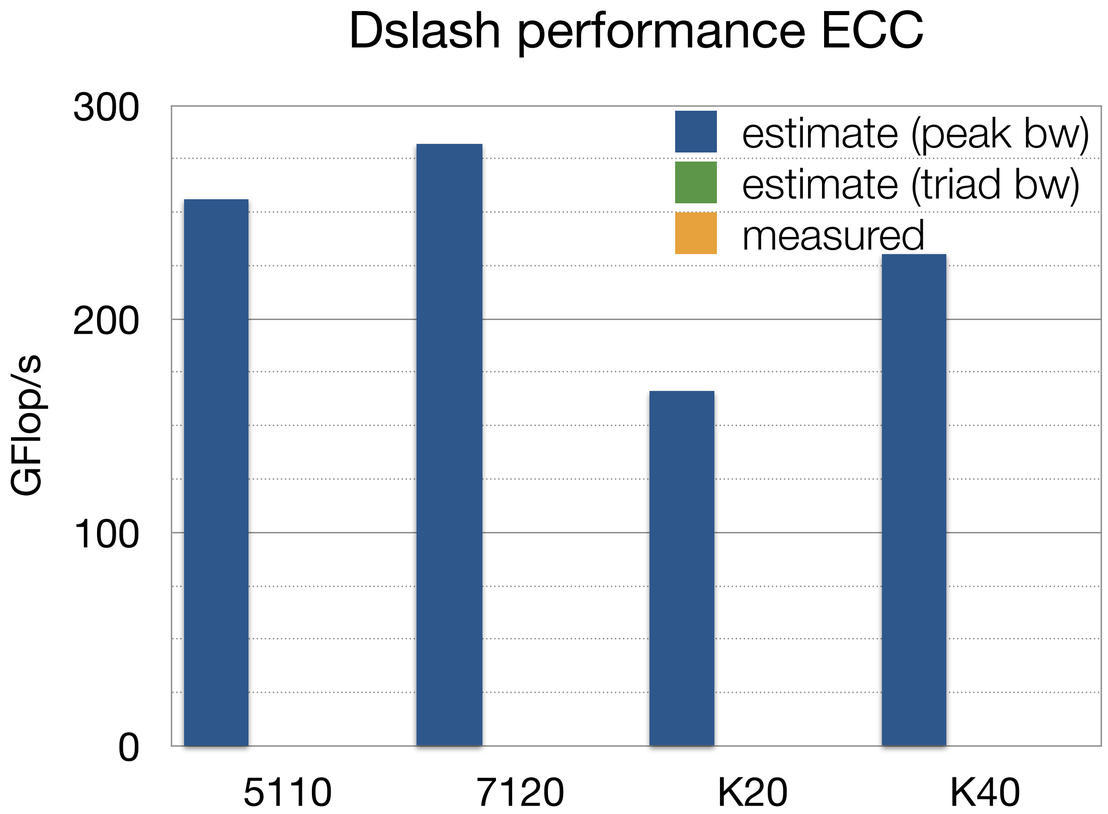


Estimated Dslash Performance

- naive model:
  bandwidth times arithmetic intensity

- better use STREAM triad bandwidth

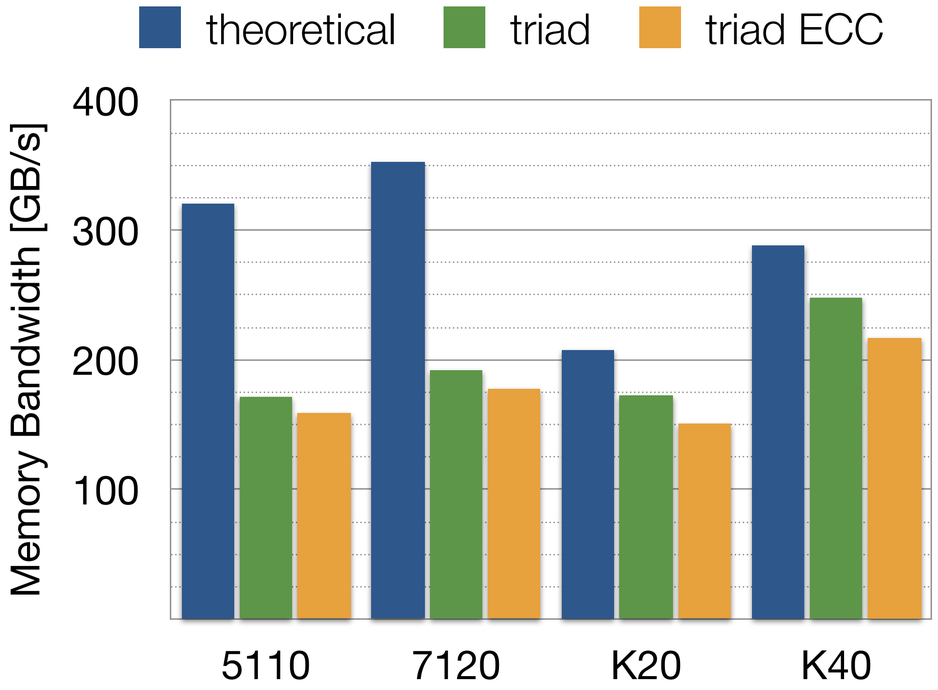


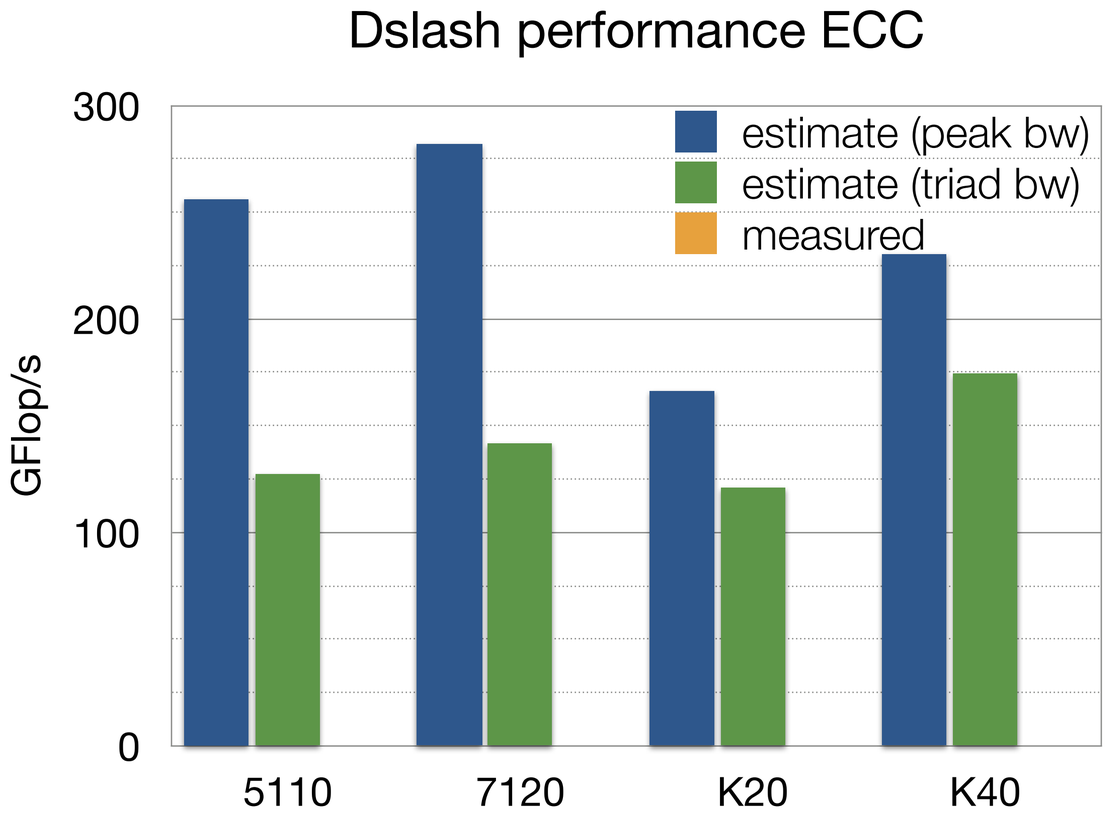


Estimated Dslash Performance

- naive model:
  bandwidth times arithmetic intensity

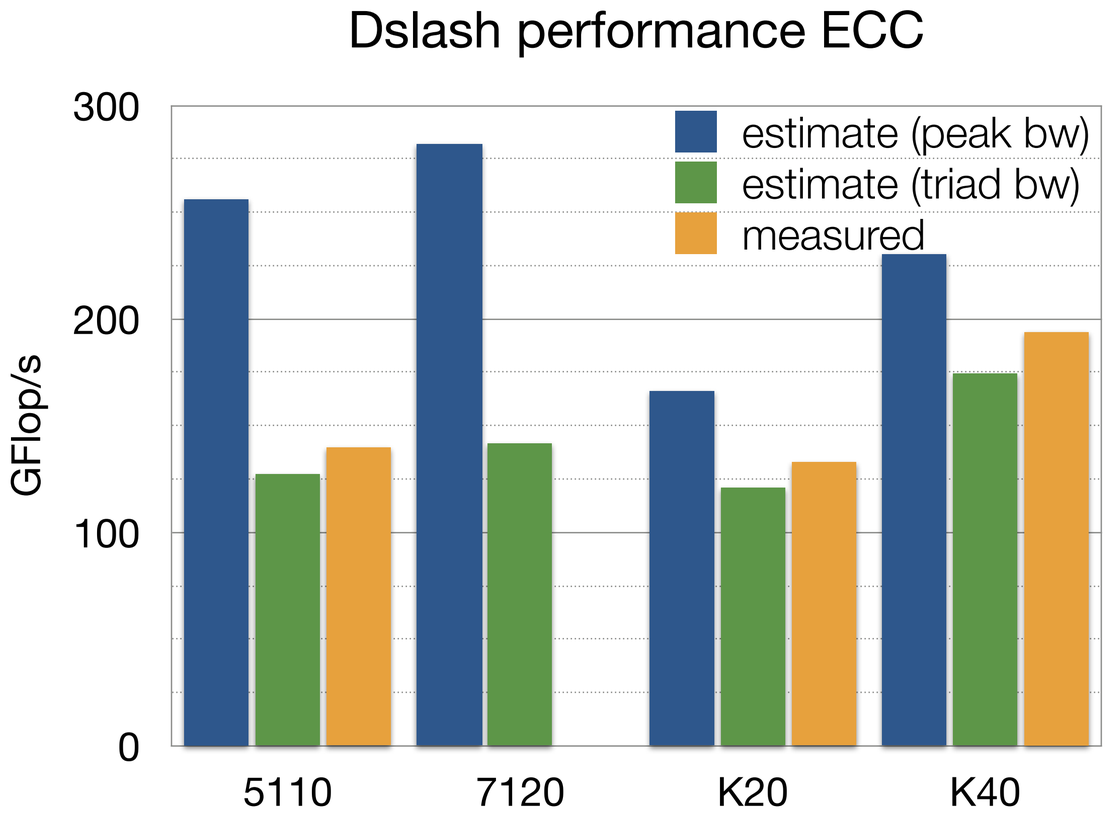
- better use STREAM triad bandwidth





Estimated Dslash Performance

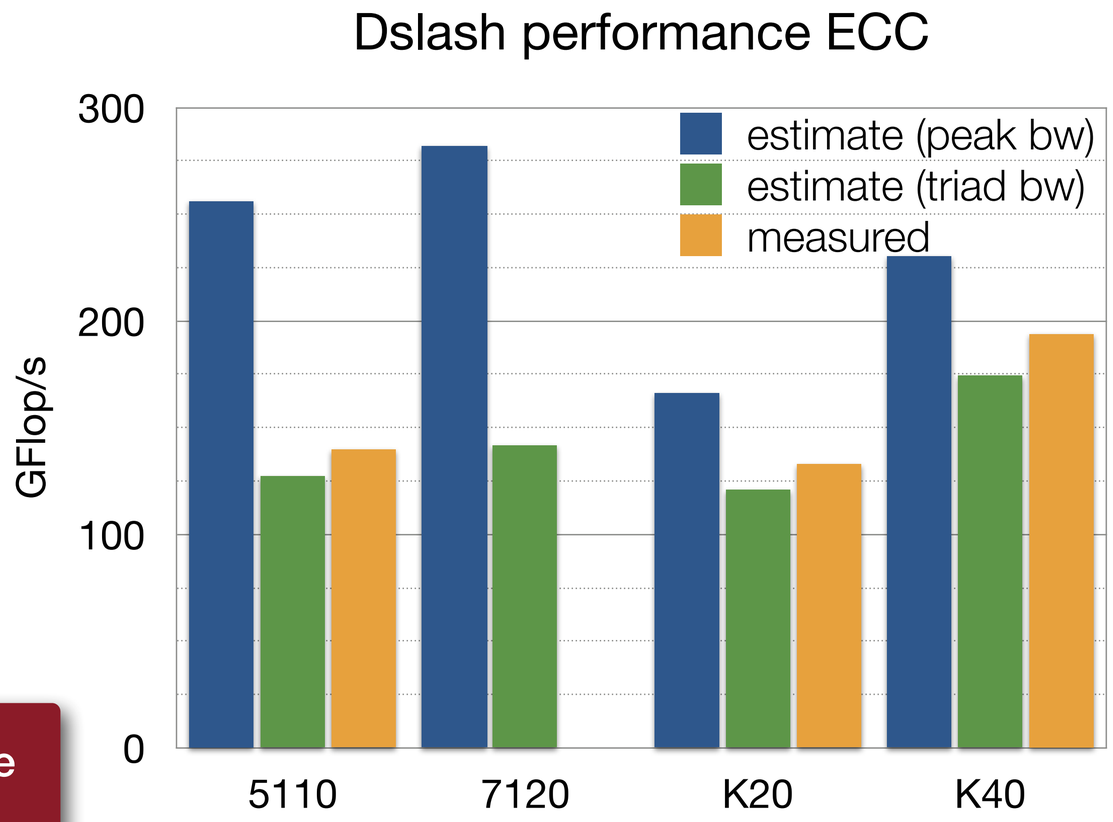
- naive model:
  bandwidth times arithmetic intensity
- better use STREAM triad bandwidth
- faster than estimate from triad bandwidth



Estimated Dslash Performance

- naive model:
  bandwidth times arithmetic intensity
- better use STREAM triad bandwidth
- faster than estimate from triad bandwidth

account for existence of cache in estimate of performance



Caching for vectors

- for upper limit: assume cache hits are free
  bytes / site: 1024 x (1-hitrate) 384 + 24

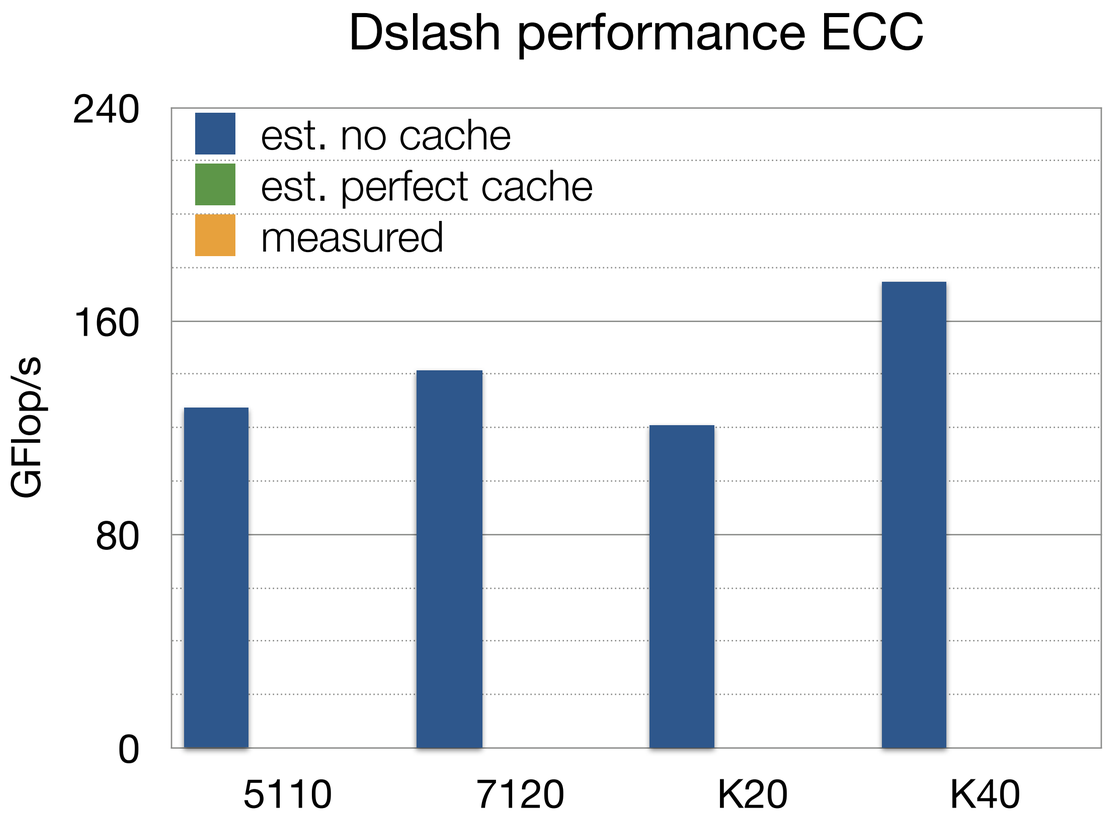
Caching for vectors

- for upper limit: assume cache hits are free
  bytes / site: 1024 x (1-hitrate) 384 + 24

gauge field

16 vectors
24 byte each

1 vectors
output



Caching for vectors

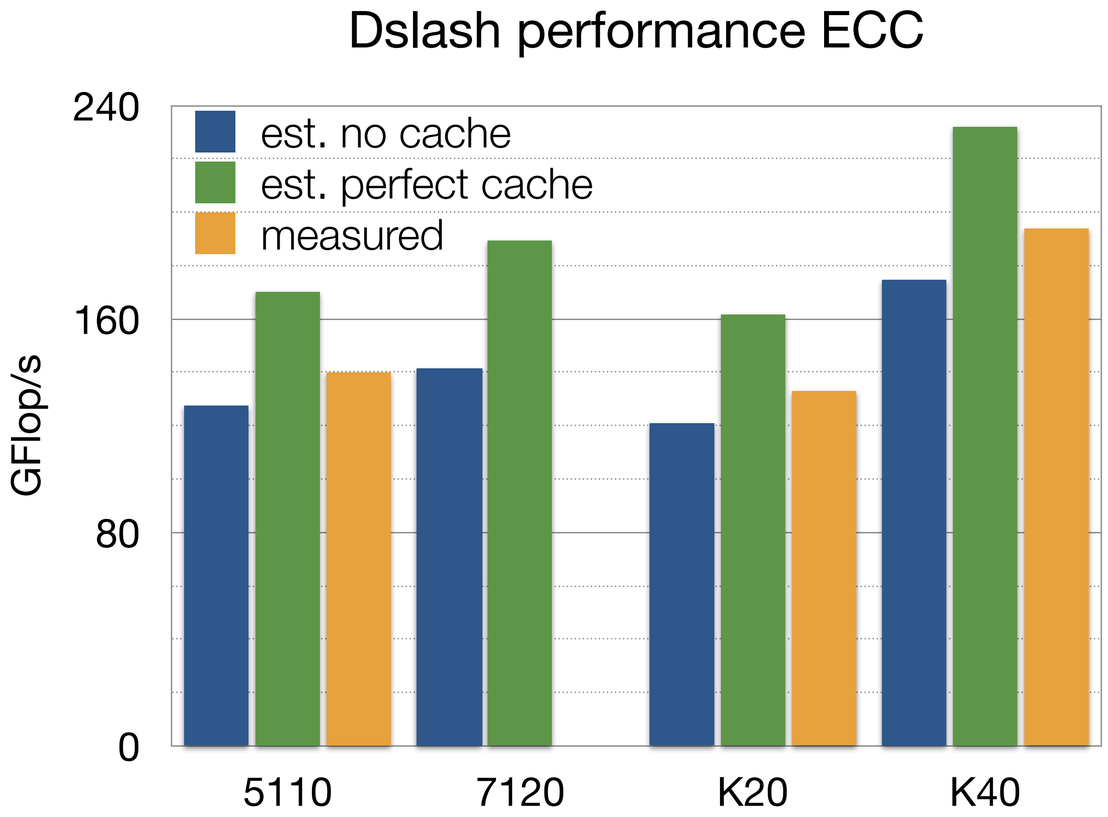
- for upper limit: assume cache hits are free
  bytes / site: 1024 x (1-hitrate) 384 + 24

gauge field

16 vectors
24 byte each

1 vectors
output

- Perfect caching scenario: hit for 15 out of 16 input vectors
  → arithmetic intensity 1.07 (w/o cache 0.80)



Caching for vectors

- for upper limit: assume cache hits are free
  bytes / site: 1024 x (1-hitrate) 384 + 24

gauge field

16 vectors
24 byte each

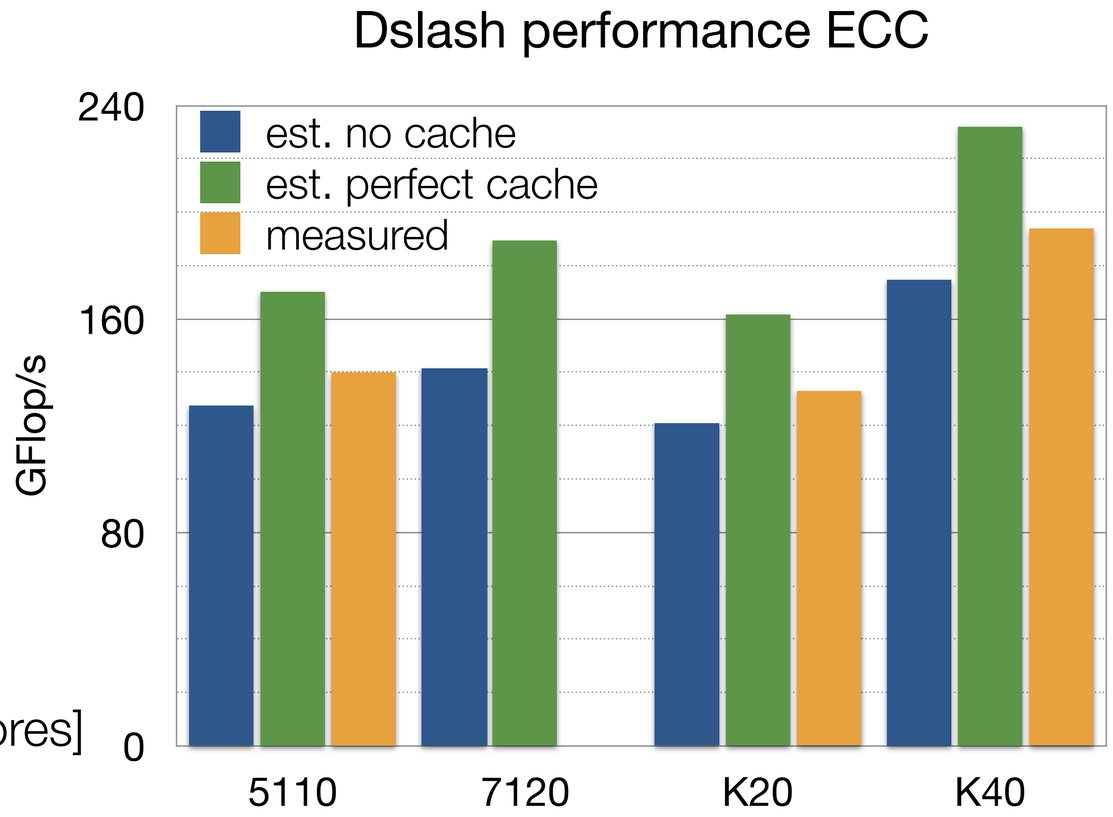
1 vectors
output

- Perfect caching scenario: hit for 15 out of 16 input vectors
  → arithmetic intensity 1.07 (w/o cache 0.80)

- typical size of a vector: 32^3x8 → 3MB, 64^3x16 → 24MB

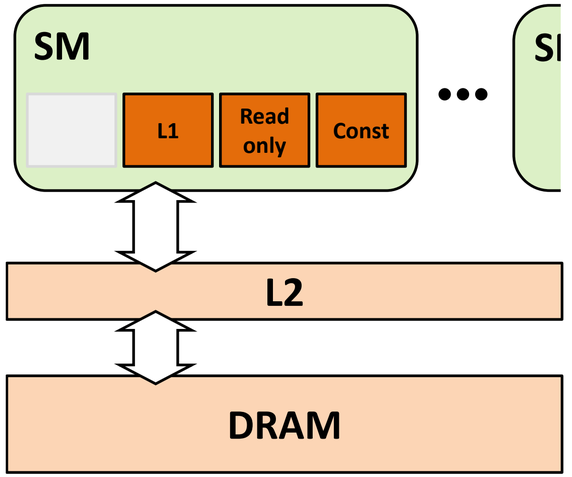
- KNL: ~30 MB L2 (512 kB / core) + 32kB L1 / core [60 cores]

- Kepler: 1.5MB L2+ (16-48) kB L1 / SMX [15 SMX]



Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)

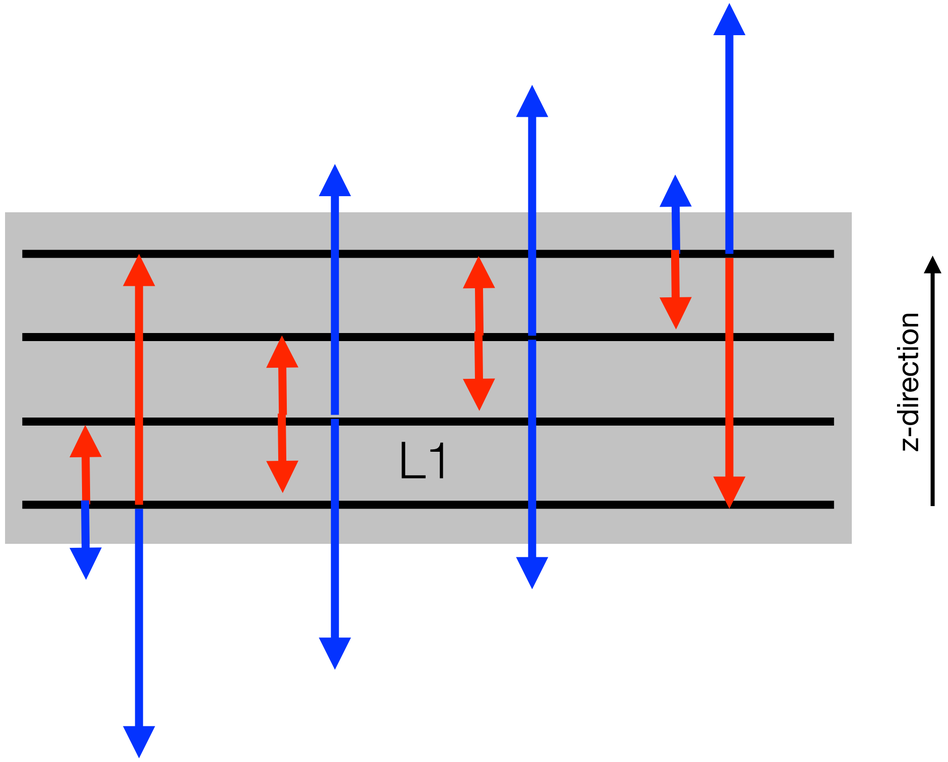


Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)

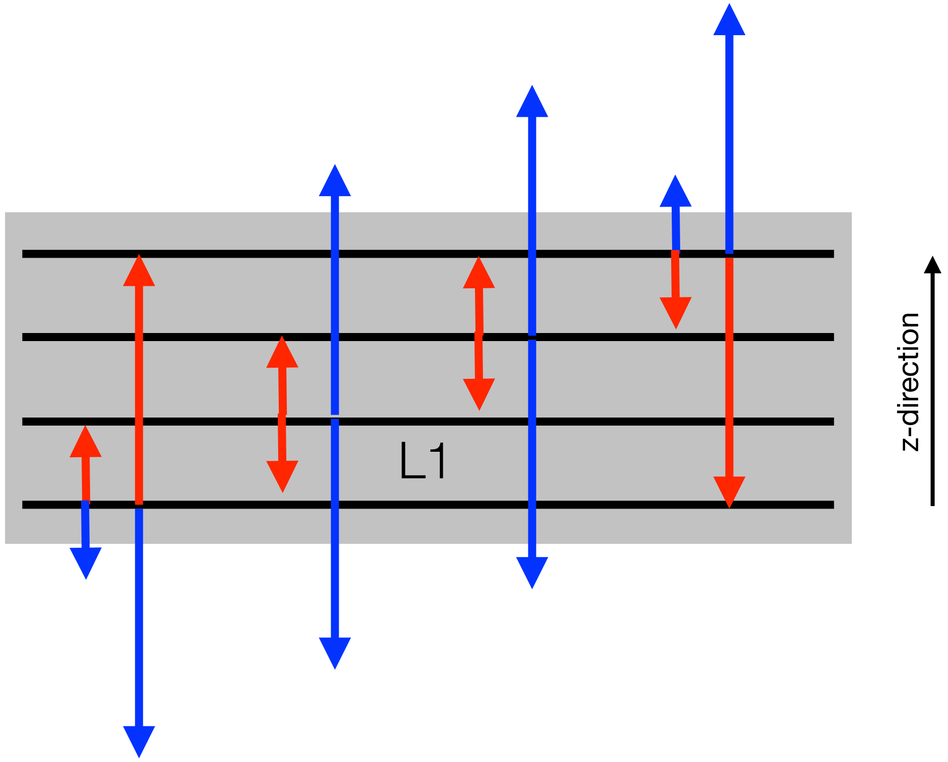
Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)



Try to get a better estimate (GPU focussed)

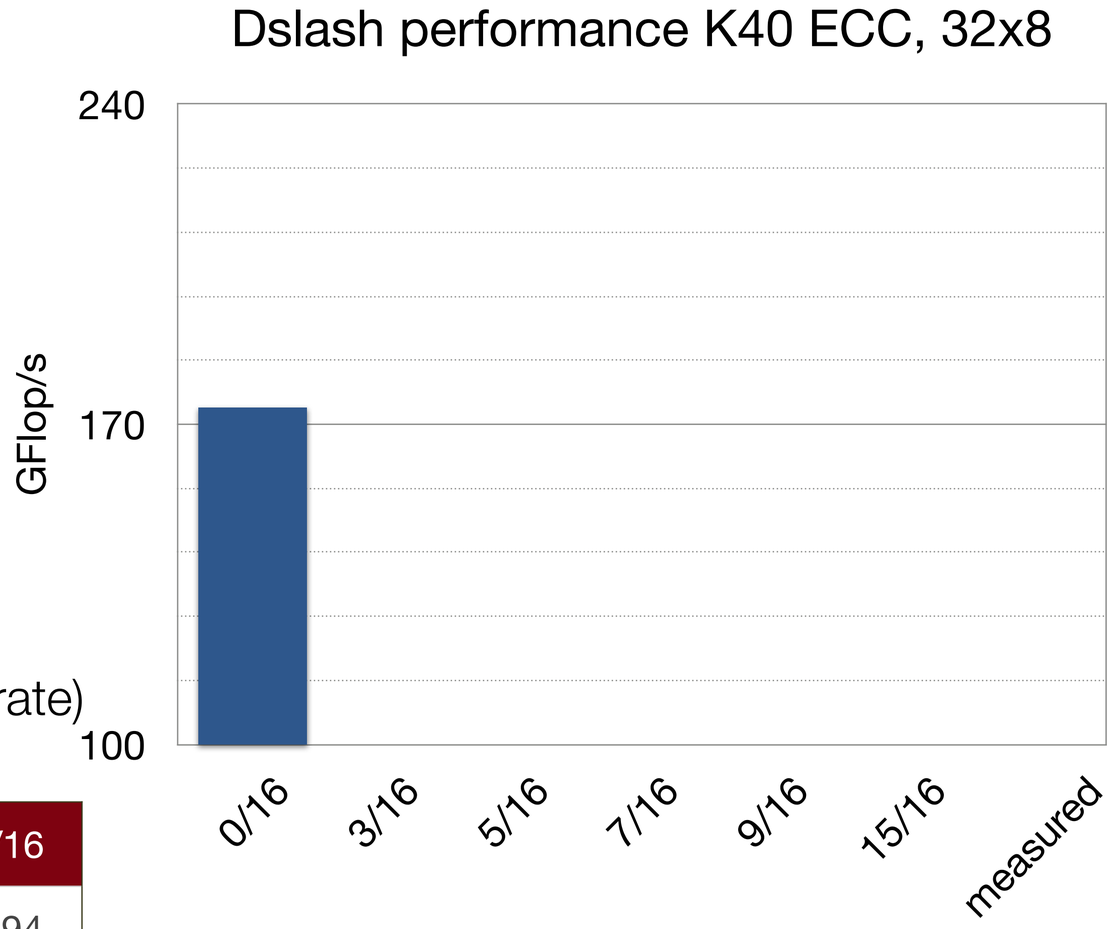
- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)



hit rate	0/16	15/16	3/16	5/16	7/16	9/16
arithmetic intensity	0.8	1.07	0.84	0.87	0.91	0.94

Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)

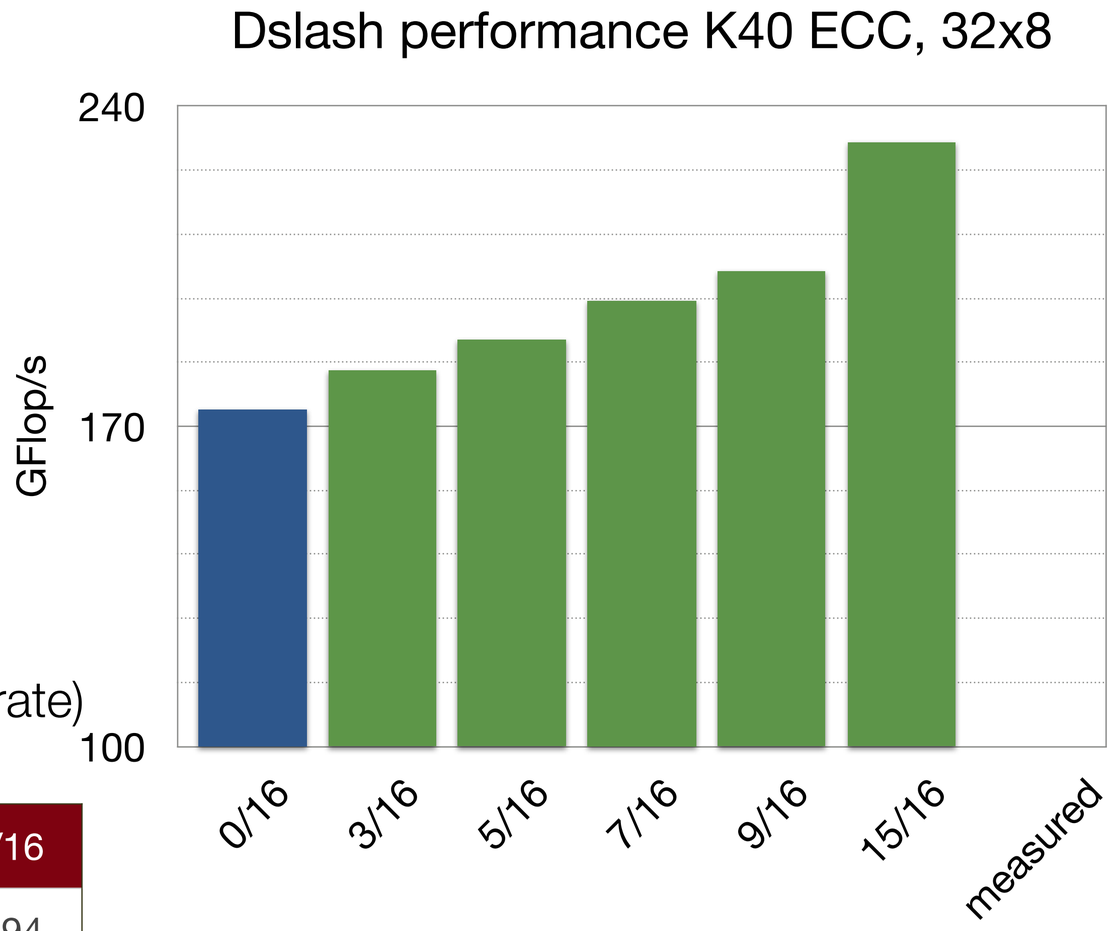


hit rate	0/16	15/16	3/16	5/16	7/16	9/16
arithmetic intensity	0.8	1.07	0.84	0.87	0.91	0.94

Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)

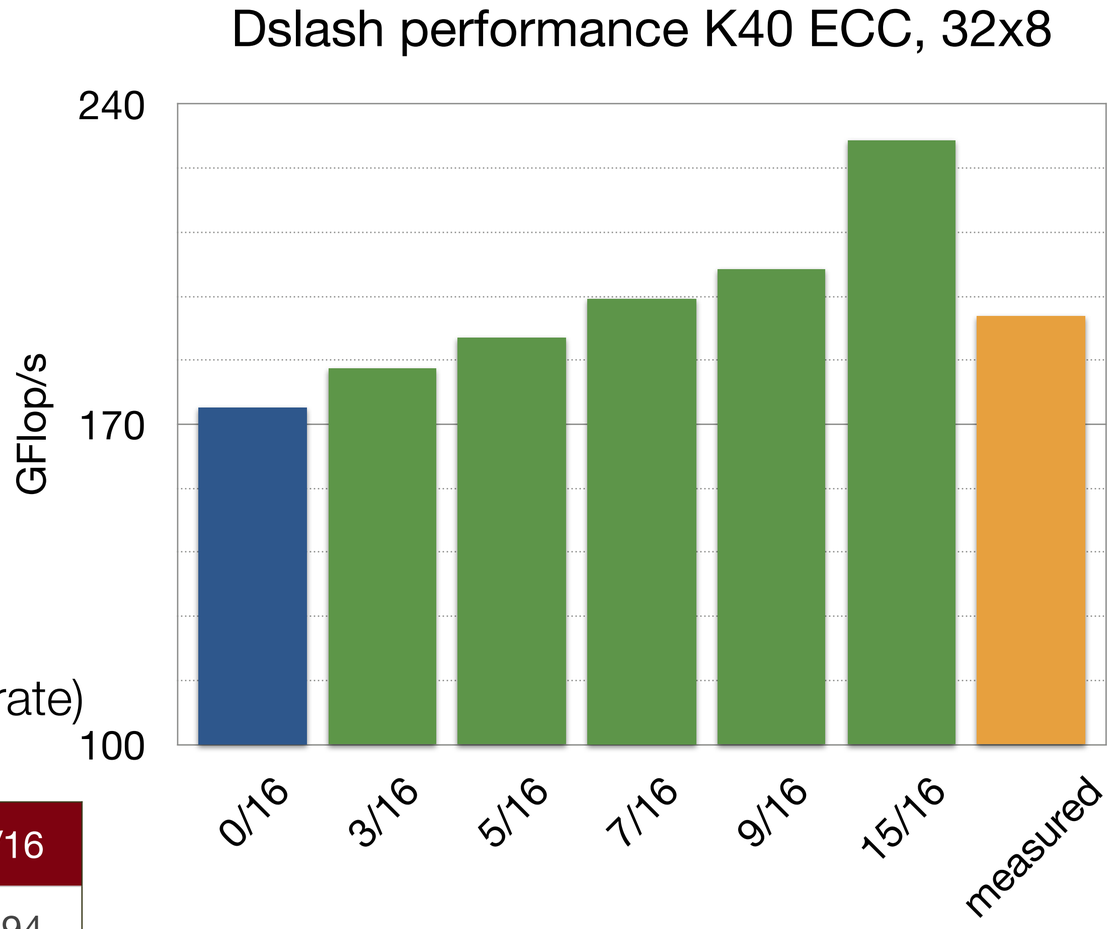
hit rate	0/16	15/16	3/16	5/16	7/16	9/16
arithmetic intensity	0.8	1.07	0.84	0.87	0.91	0.94



Try to get a better estimate (GPU focussed)

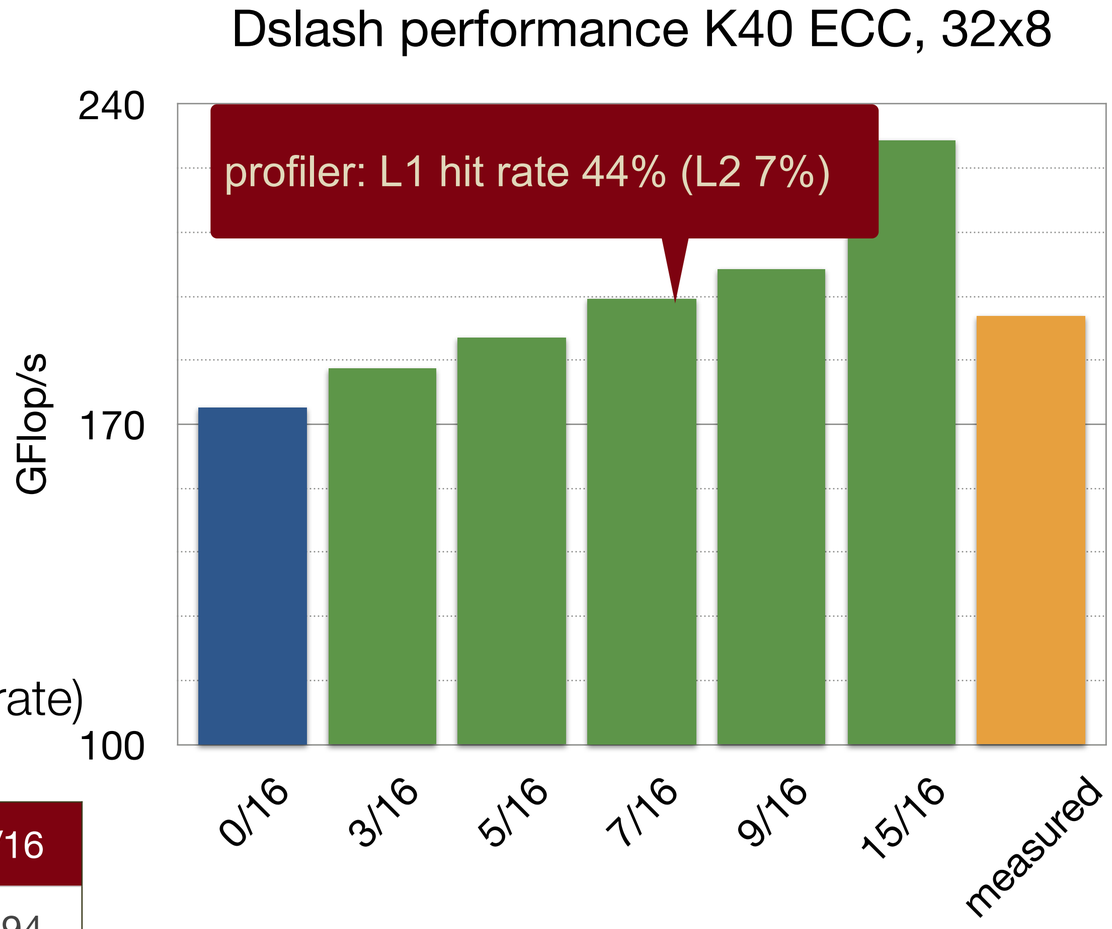
- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)

hit rate	0/16	15/16	3/16	5/16	7/16	9/16
arithmetic intensity	0.8	1.07	0.84	0.87	0.91	0.94



Try to get a better estimate (GPU focussed)

- Empirical: vectors through L1, links through texture
- ignore L2: also loads gauge field (128MB - 1024MB)
- 48 kB L1 can hold 2048 24-byte vector elements
  - for 64^3x16: 1 xy-plane (even-odd precondition)
    hit 7 out of 16 (43% hit rate)
  - for 32^3x8: xy plane has 512 elements → 4 xy-planes
    in z direction we can hit 2 of 4 elements: 9/16 (56% hit rate)



hit rate	0/16	15/16	3/16	5/16	7/16	9/16
arithmetic intensity	0.8	1.07	0.84	0.87	0.91	0.94

Increasing the Intensity

Focus on the arithmetic intensity now … push ups later.

Cache effects for vectors but remember they are only ~25% of the memory traffic.

What can we do about the gauge links ?

HISQ Inverter for multiple right hand sides (rhs)

- combine multiple inversions with constant gauge field (constant sparse matrix)

$$\left(w_x^{(1)}, w_x^{(2)}, \ldots, w_x^{(n)}\right) = D_{x,x'}\left(v_{x'}^{(1)}, v_{x'}^{(2)}, \ldots, v_x^{(n)}\right)$$

- reuse links (input for the sparse matrix) in the matrix-vector multiplication (Dslash)

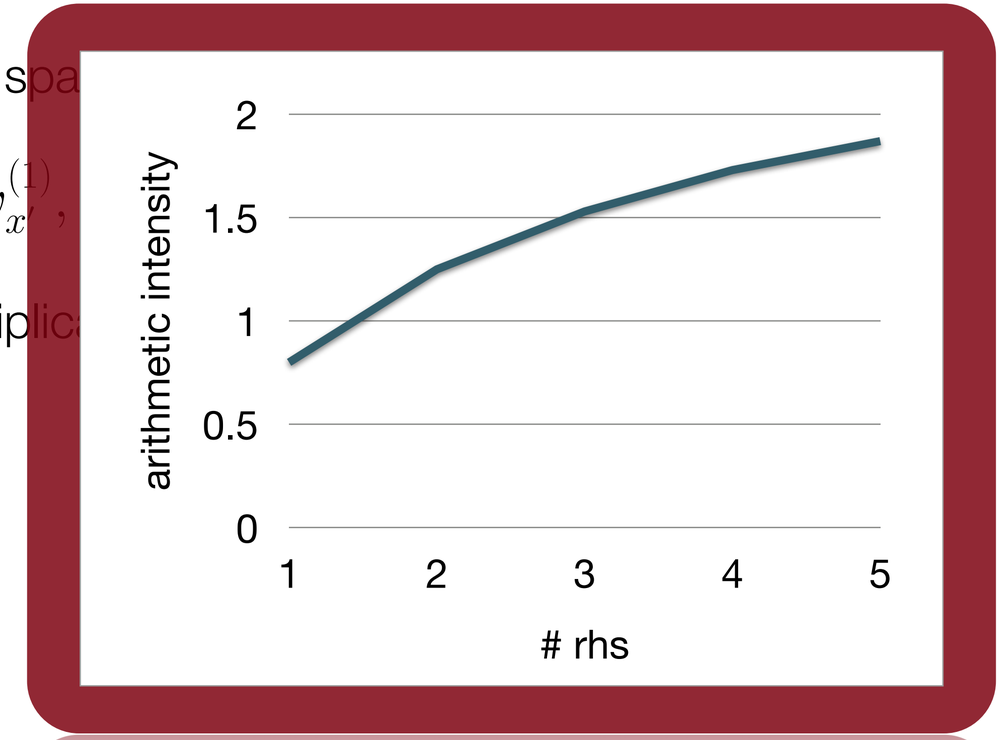
HISQ Inverter for multiple right hand sides (rhs)

- combine multiple inversions with constant gauge field (constant spa

$$\left(w_x^{(1)}, w_x^{(2)}, \ldots, w_x^{(n)}\right) = D_{x,x'}\left(v_{x'}^{(1)},\right.$$

- reuse links (input for the sparse matrix) in the matrix-vector multiplica

#rhs	1	2	3	4	5
Flop/byte	0.80	1.25	1.53	1.73	1.87



HISQ Inverter for multiple right hand sides (rhs)

- combine multiple inversions with constant gauge field (constant sparse matrix)

$$\left(w_x^{(1)}, w_x^{(2)}, \ldots, w_x^{(n)}\right) = D_{x,x'} \left(v_{x'}^{(1)}, v_{x'}^{(2)}, \ldots, v_x^{(n)}\right)$$

- reuse links (input for the sparse matrix) in the matrix-vector multiplication (Dslash)

#rhs	1	2	3	4	5
Flop/byte	0.80	1.25	1.53	1.73	1.87

- ignored cache effects for vectors here
  - caching will be much harder now as cache needs to be shared by vectors for #rhs
- memory traffic from gauge links decreases from 70% (1 rhs) to 30% (4 rhs)

GPU Implementation: Texture Cache and Registers

- obvious solution: store matrix in registers
  - possible issue: more registers / thread
    → occupancy / spilling

GPU Implementation: Texture Cache and Registers

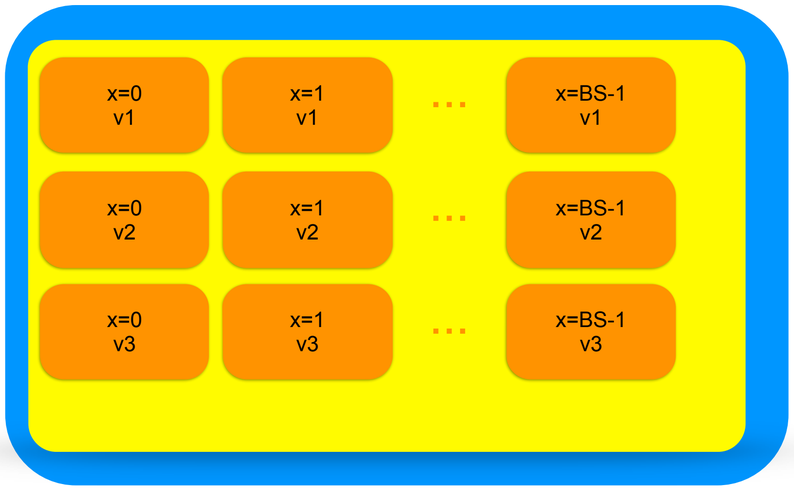
- obvious solution: store matrix in registers
  - possible issue: more registers / thread
    → occupancy / spilling

```
__global__ Dslashreg (w1, w2, w3, v1, v2, v3 ){
...
for(xp=...){
    w1(x) = D(x,xp) * v1(xp);
    w2(x) = D(x,xp) * v2(xp);
    w3(x) = D(x,xp) * v3(xp);
    }
}
```

GPU Implementation: Texture Cache and Registers

- obvious solution: store matrix in registers
  - possible issue: more registers / thread
    → occupancy / spilling
- exploit texture cache
  → reduce register pressure
  - links should hit in texture cache
    → only one global load
  - one block is executed by one SMX

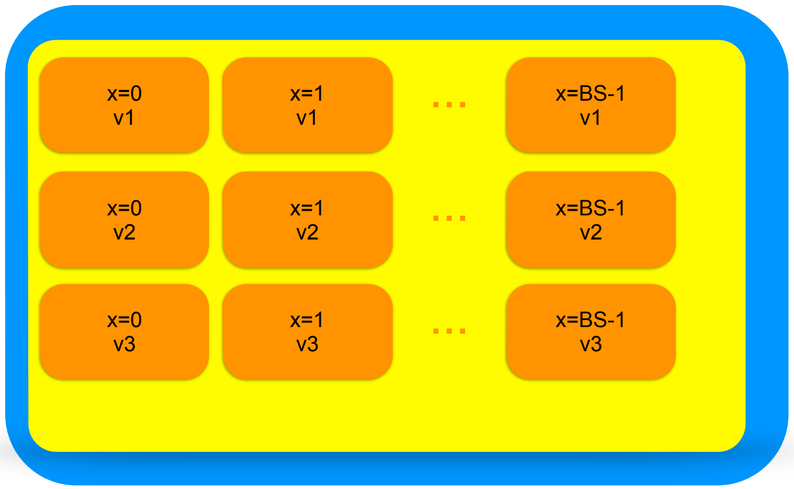
```
__global__ Dslashcache (w, v)
...
offset = threadIdx.y;
for(xp=...)
    w(x, offset) += D(x,xp) * v(x, offset)
}
```



GPU Implementation: Texture Cache and Registers

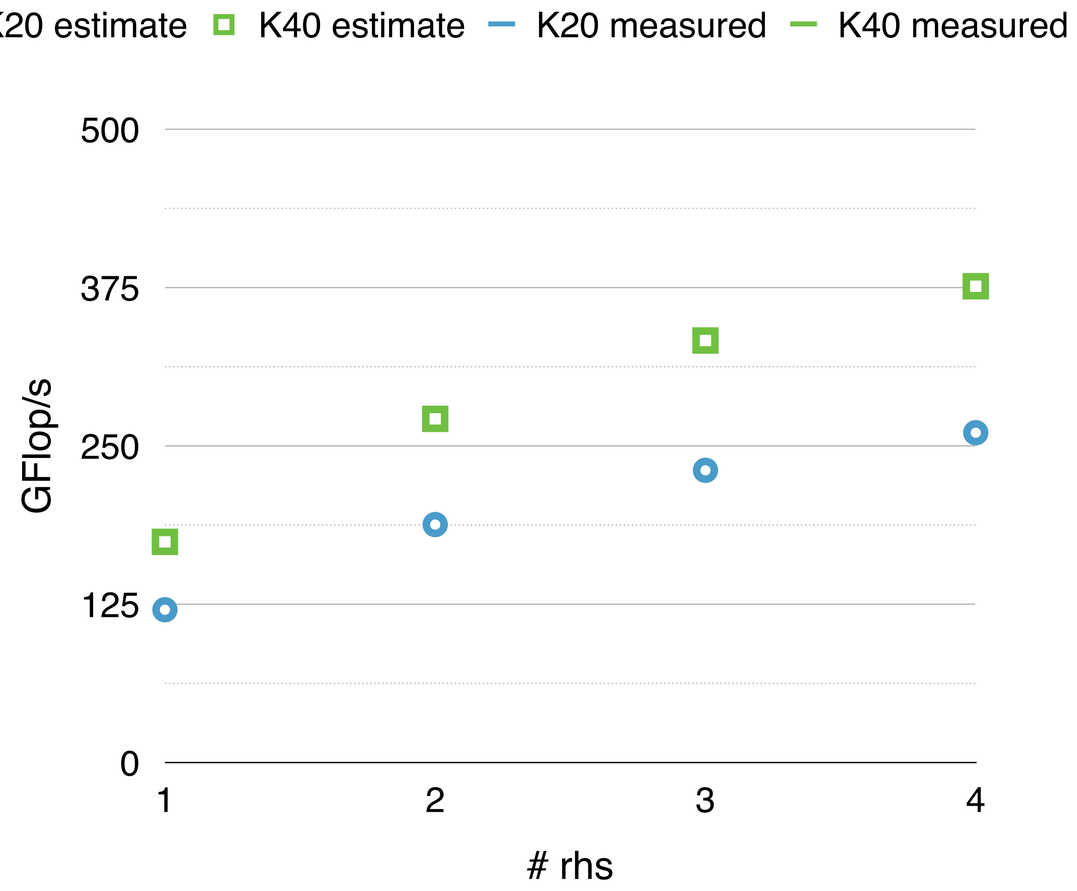
- obvious solution: store matrix in registers
  - possible issue: more registers / thread
    → occupancy / spilling
- exploit texture cache
  → reduce register pressure
  - links should hit in texture cache
    → only one global load
  - one block is executed by one SMX
- combine both and explore best possible combinations

```
__global__ Dslashregcache (w1, w2, w3, v1, v2, v3 ){
...
offset = threadIdx.y;
for(xp=...){
    w1(x, offset) = D(x,xp) * v1(xp, offset);
    w2(x, offset) = D(x,xp) * v2(xp, offset);
    w3(x, offset) = D(x,xp) * v3(xp, offset);
    }
}
```



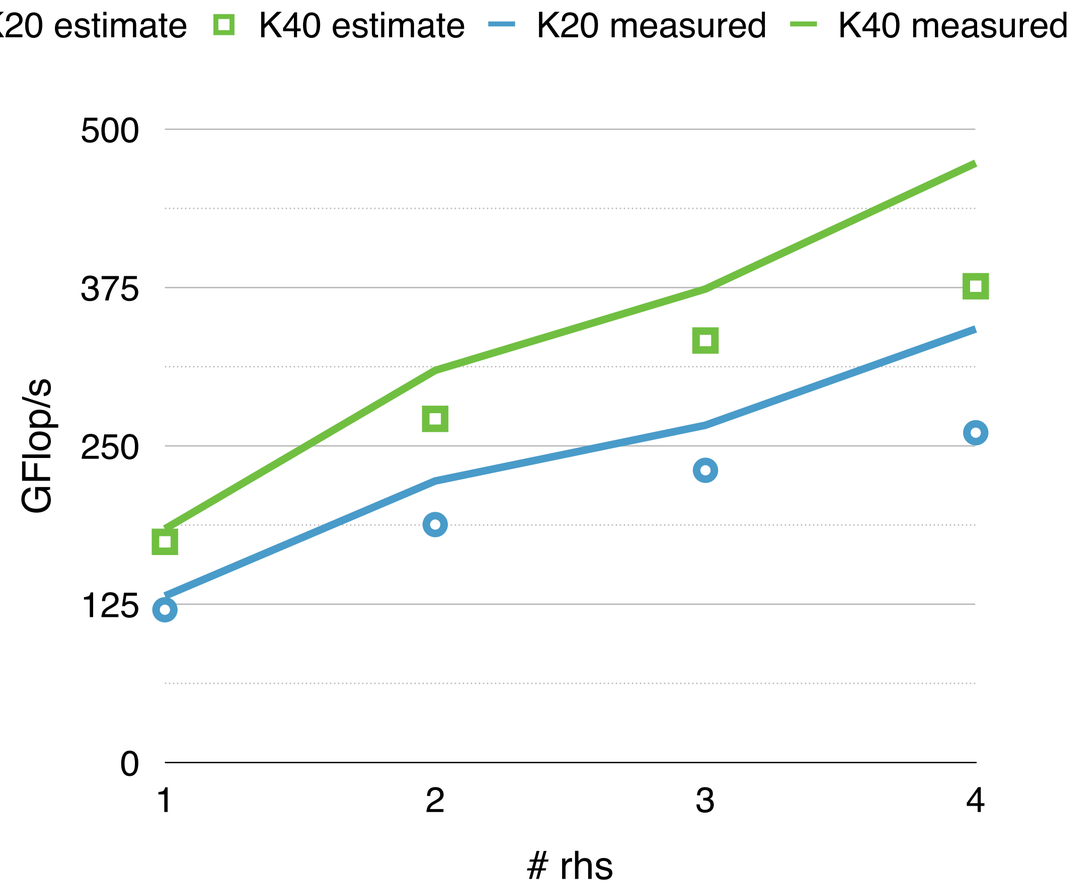
Does it work ?

- use only memory bandwidth and arithmetic intensity
- estimate with bandwidth from triad benchmark



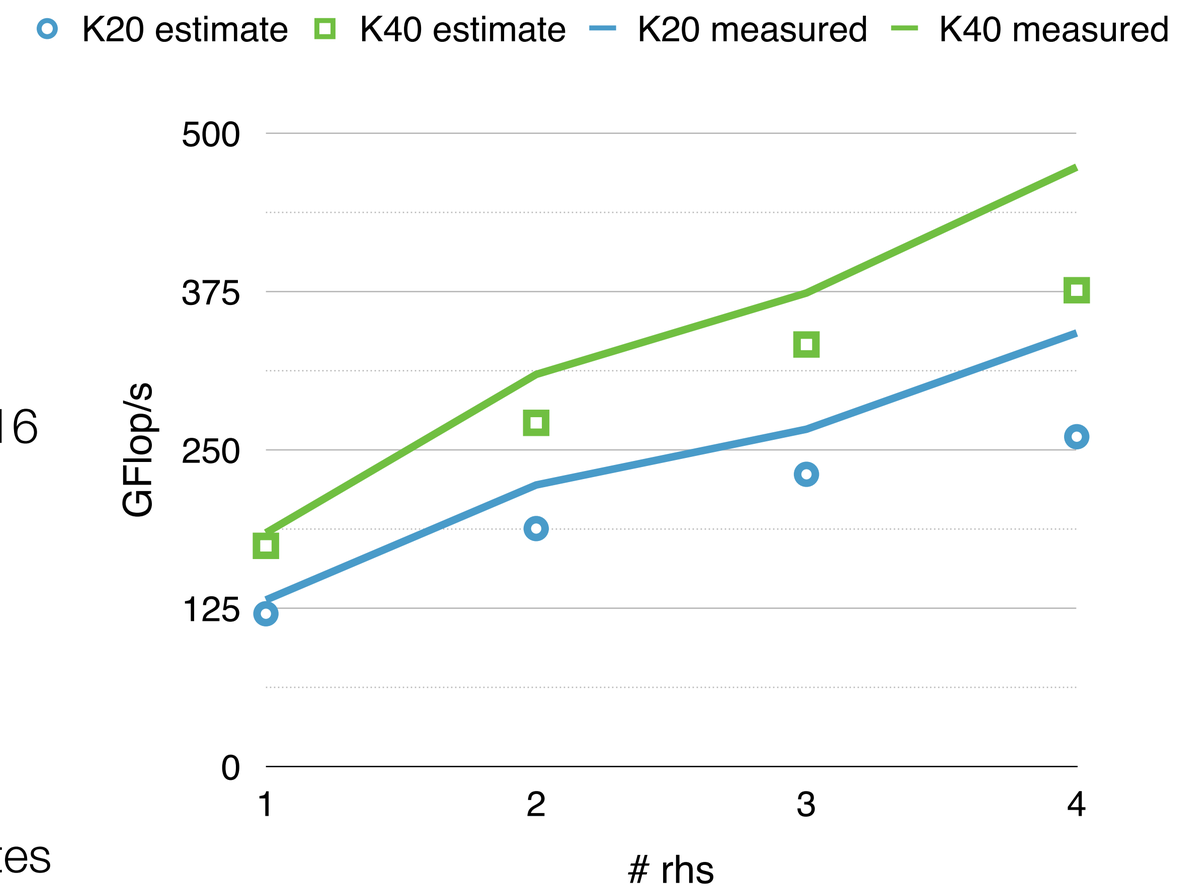
Does it work ?

- use only memory bandwidth and arithmetic intensity
- estimate with bandwidth from triad benchmark
- works even better than expected
  - expectation speedup for 4 rhs / 1 rhs:1.73/0.8 ~ 2.16
  - observed speedup: ~ 2.5
    - makes more efficient use of GPU (why ?)



Does it work ?

- use only memory bandwidth and arithmetic intensity
- estimate with bandwidth from triad benchmark
- works even better than expected
  - expectation speedup for 4 rhs / 1 rhs:1.73/0.8 ~ 2.16
  - observed speedup: ~ 2.5
    - makes more efficient use of GPU (why ?)
- pure loading through texture cache always wins
  - but 48kB texture cache can only hold links for 48 sites (each sites need 8x72 bytes + 8x56 bytes)



Ask the profiler

- profile for 4 rhs to see whether caching strategy works:

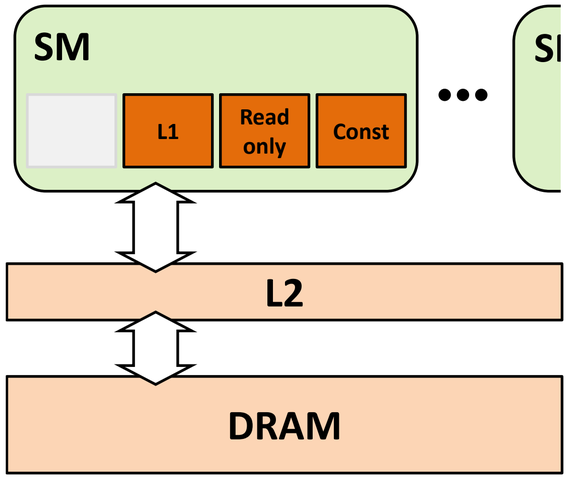
Block	regs	occup.	eligibl. warps	IPC
[16,4]	63	0.49	2.45	1.92
[128,4]	63	0.47	2.92	1.92
[256,4]	63	0.48	3.08	1.87
[1024,1]	62	0.48	0.87	0.77

Ask the profiler

- profile for 4 rhs to see whether caching strategy works:

Block	regs	occup.	eligibl. warps	IPC	TC Hits %	L2 (TC) Hits %
[16,4]	63	0.49	2.45	1.92	51.9	50.0
[128,4]	63	0.47	2.92	1.92	74.3	5.6
[256,4]	63	0.48	3.08	1.87	75.9	0.0
[1024,1]	62	0.48	0.87	0.77	3.8	0.0

- each gauge link loaded once / rhs → best case 75% texture cache hit

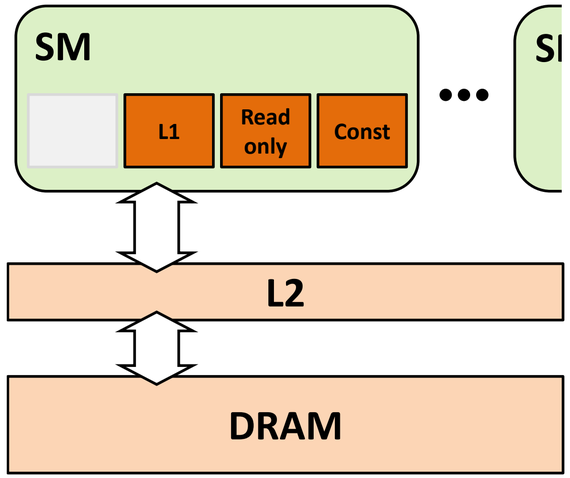


Ask the profiler

- profile for 4 rhs to see whether caching strategy works:

Block	regs	occup.	eligibl. warps	IPC	TC Hits %	L2 (TC) Hits %	L1 Hits %	L2 (L1) Hits %
[16,4]	63	0.49	2.45	1.92	51.9	50.0	18.2	48.4
[128,4]	63	0.47	2.92	1.92	74.3	5.6	31.2	37.1
[256,4]	63	0.48	3.08	1.87	75.9	0.0	33.9	28.9
[1024,1]	62	0.48	0.87	0.77	3.8	0.0	44.3	7.1

- each gauge link loaded once / rhs → best case 75% texture cache hit



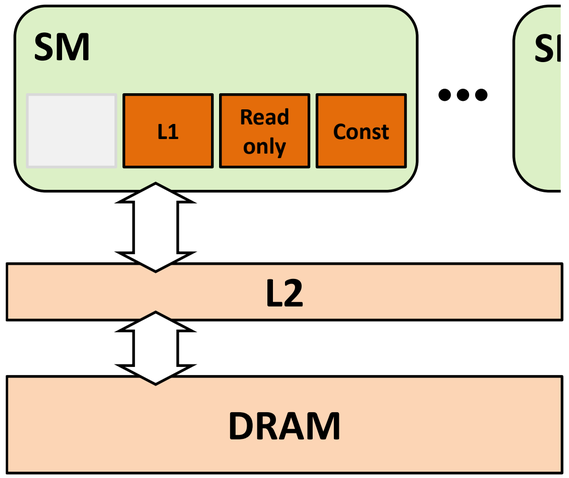
Ask the profiler

- profile for 4 rhs to see whether caching strategy works:

Block	regs	occup.	eligibl. warps	IPC	TC Hits %	L2 (TC) Hits %	L1 Hits %	L2 (L1) Hits %	Tex+L2 Hits %	L1+L2 Hits %
[16,4]	63	0.49	2.45	1.92	51.9	50.0	18.2	48.4	75.9	57.8
[128,4]	63	0.47	2.92	1.92	74.3	5.6	31.2	37.1	75.7	56.7
[256,4]	63	0.48	3.08	1.87	75.9	0.0	33.9	28.9	75.9	53.0
[1024,1]	62	0.48	0.87	0.77	3.8	0.0	44.3	7.1	3.8	48.3

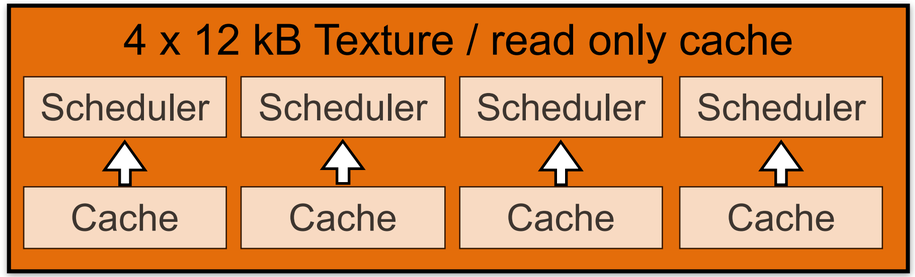
- each gauge link loaded once / rhs → best case 75% texture cache hit

- better speedup than expected for 4 rhs compared to 1 rhs:
  - better utilization of GPU and better use of L2 cache



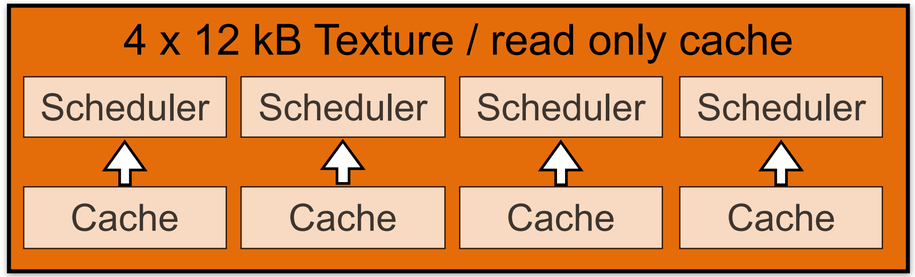
Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler



Can we understand why it works ?

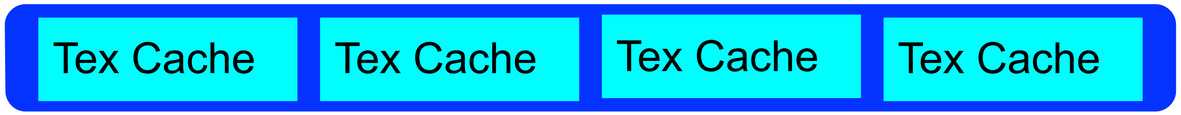
- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

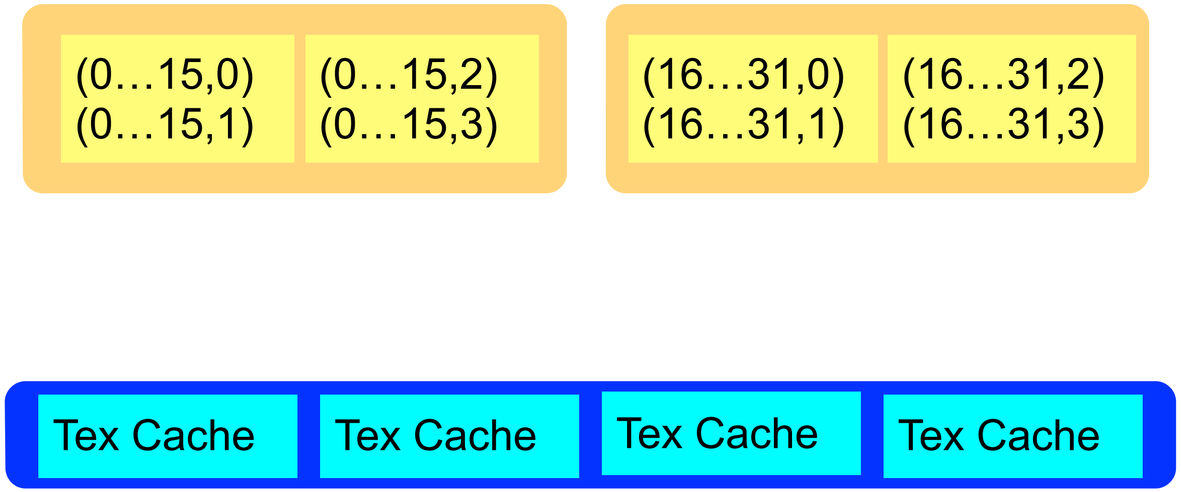
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

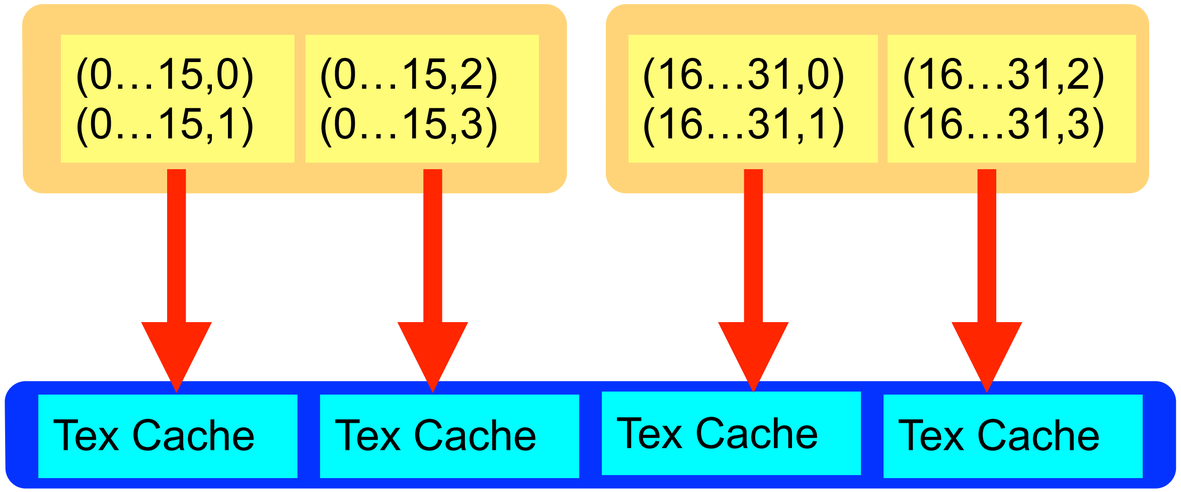
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

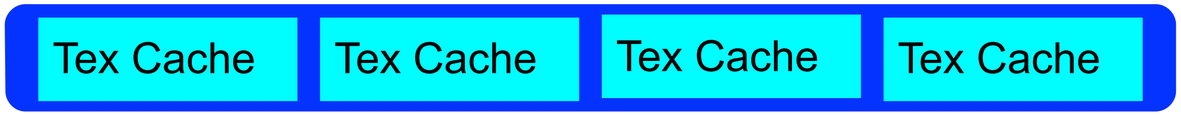
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7

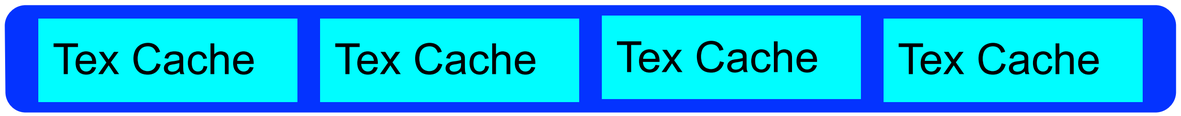


Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7

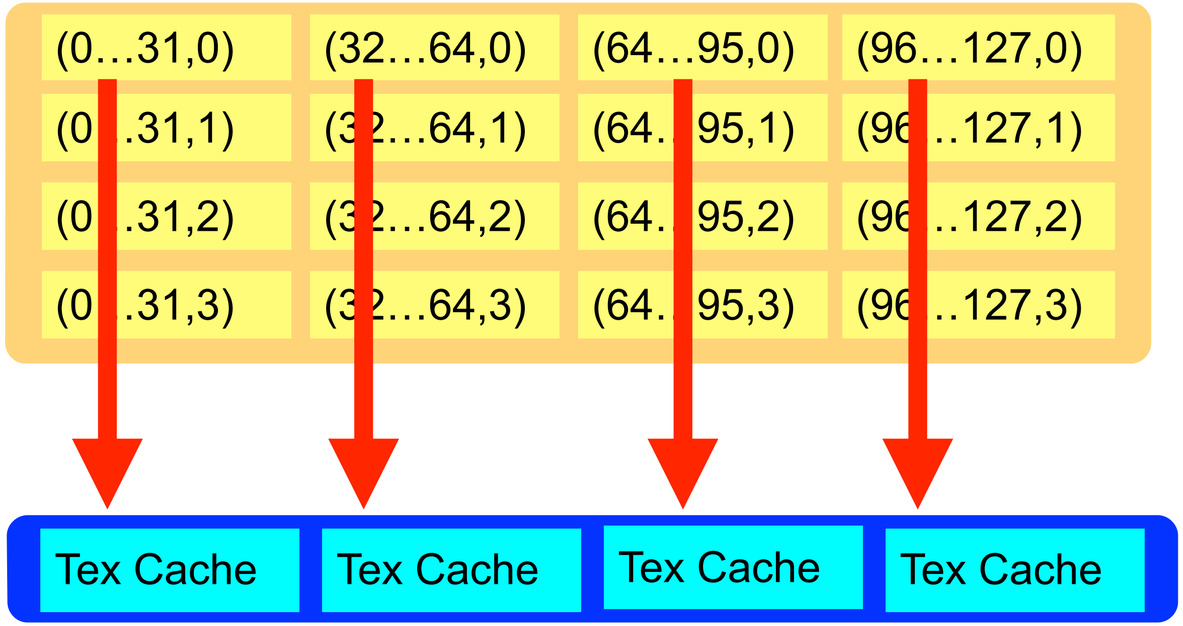
(0…31,0)	(32…64,0)	(64…95,0)	(96…127,0)
(0…31,1)	(32…64,1)	(64…95,1)	(96…127,1)
(0…31,2)	(32…64,2)	(64…95,2)	(96…127,2)
(0…31,3)	(32…64,3)	(64…95,3)	(96…127,3)



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

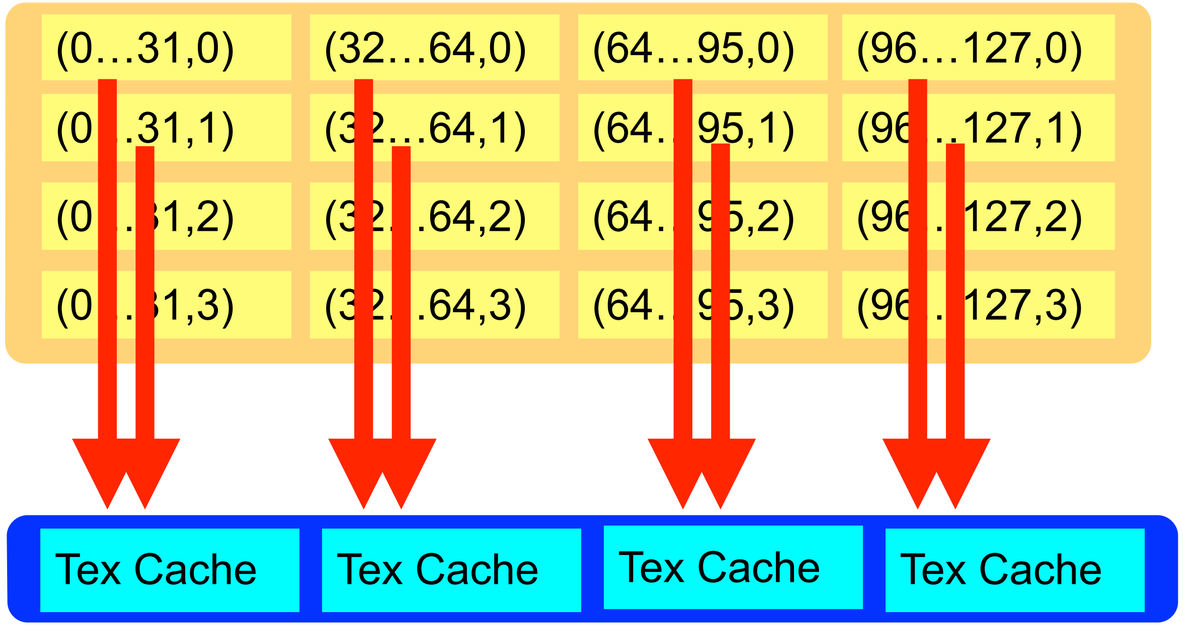
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

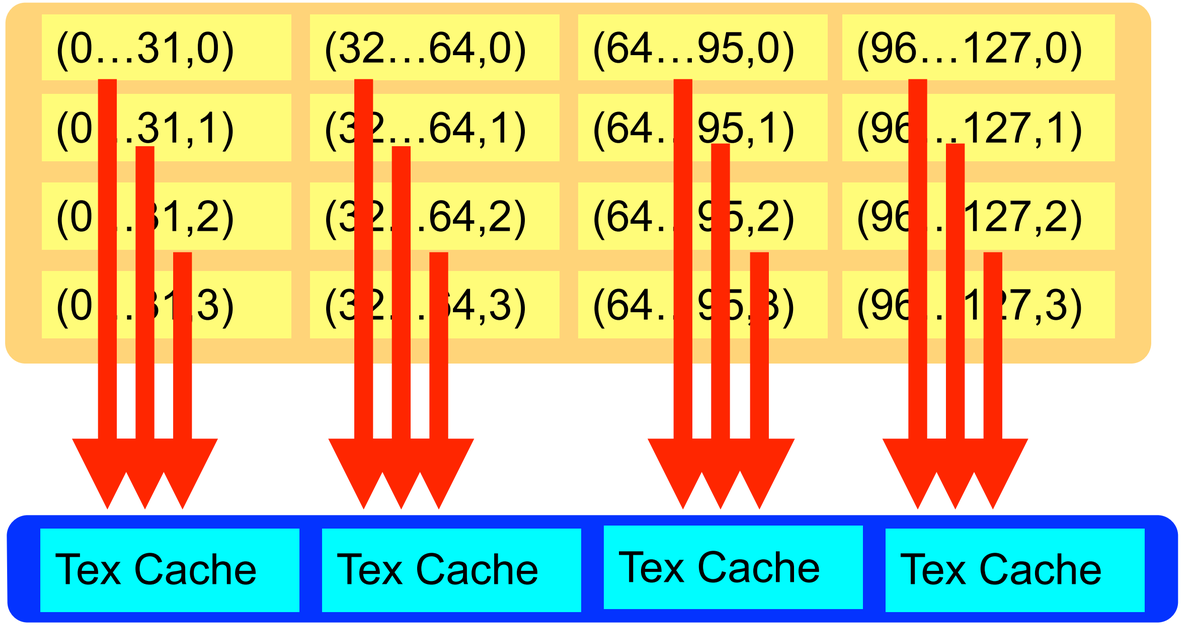
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

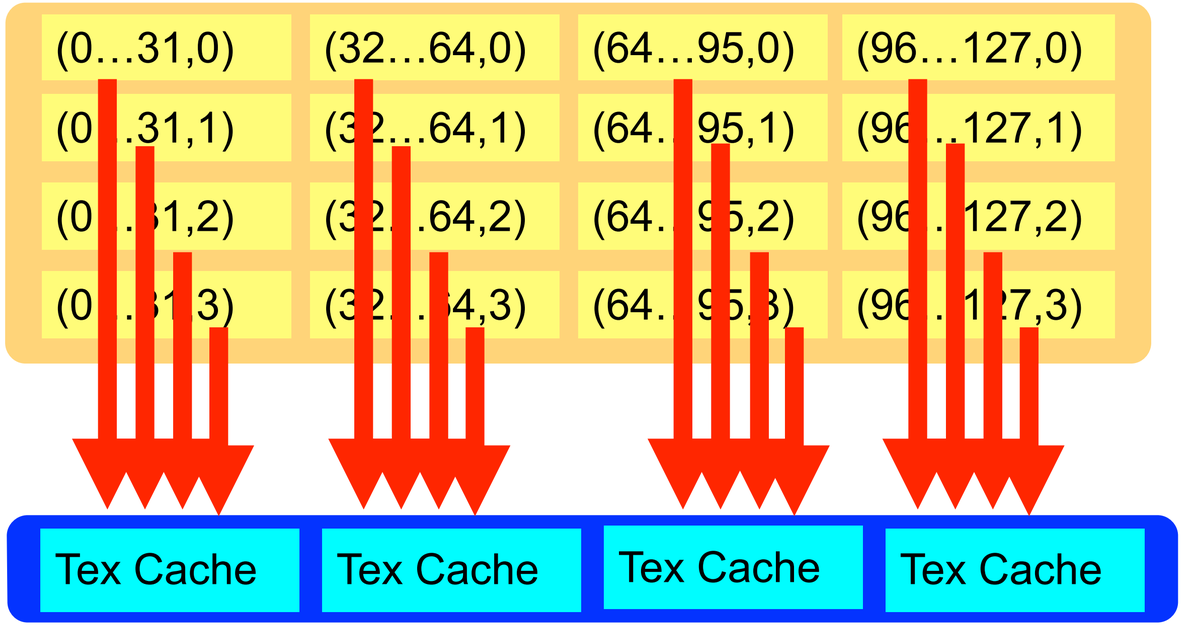
Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7



Can we understand why it works ?

- focus on pure texture cache solution [1,4]
- each thread needs (8 x 72 + 8 x 56)=1024 bytes
- warps (32 threads) assigned to one scheduler
- switching between threads: need only some of the data
- block sizes and warps
  - [16,4] → 2 warps
  - [128,4] → 16 warps

Block	TC Hits %	L2 (TC) Hits %	Tex+L2 Hits %
[16,4]	51.9	50.0	75.9
[128,4]	74.3	5.6	75.7

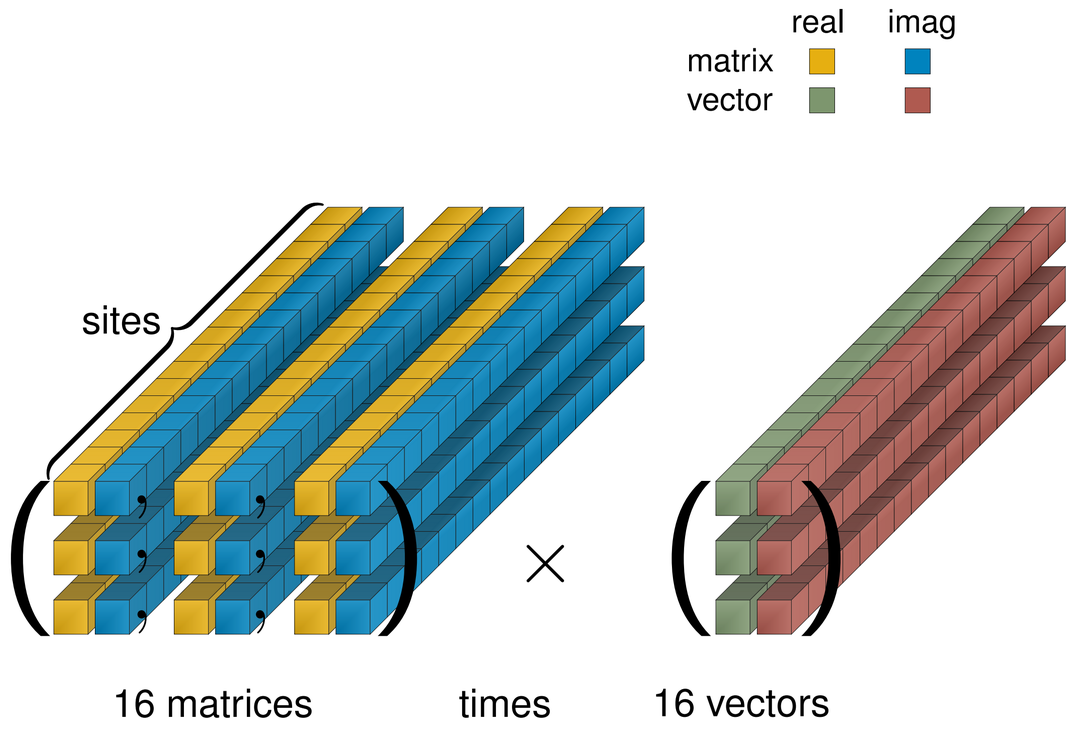


Some Details of the Phi Implementation

- effort lead by Patrick Steinbrecher (Universität Bielefeld → Brookhaven National Lab)
- single accelerator
- optimized for performance with multiple rhs

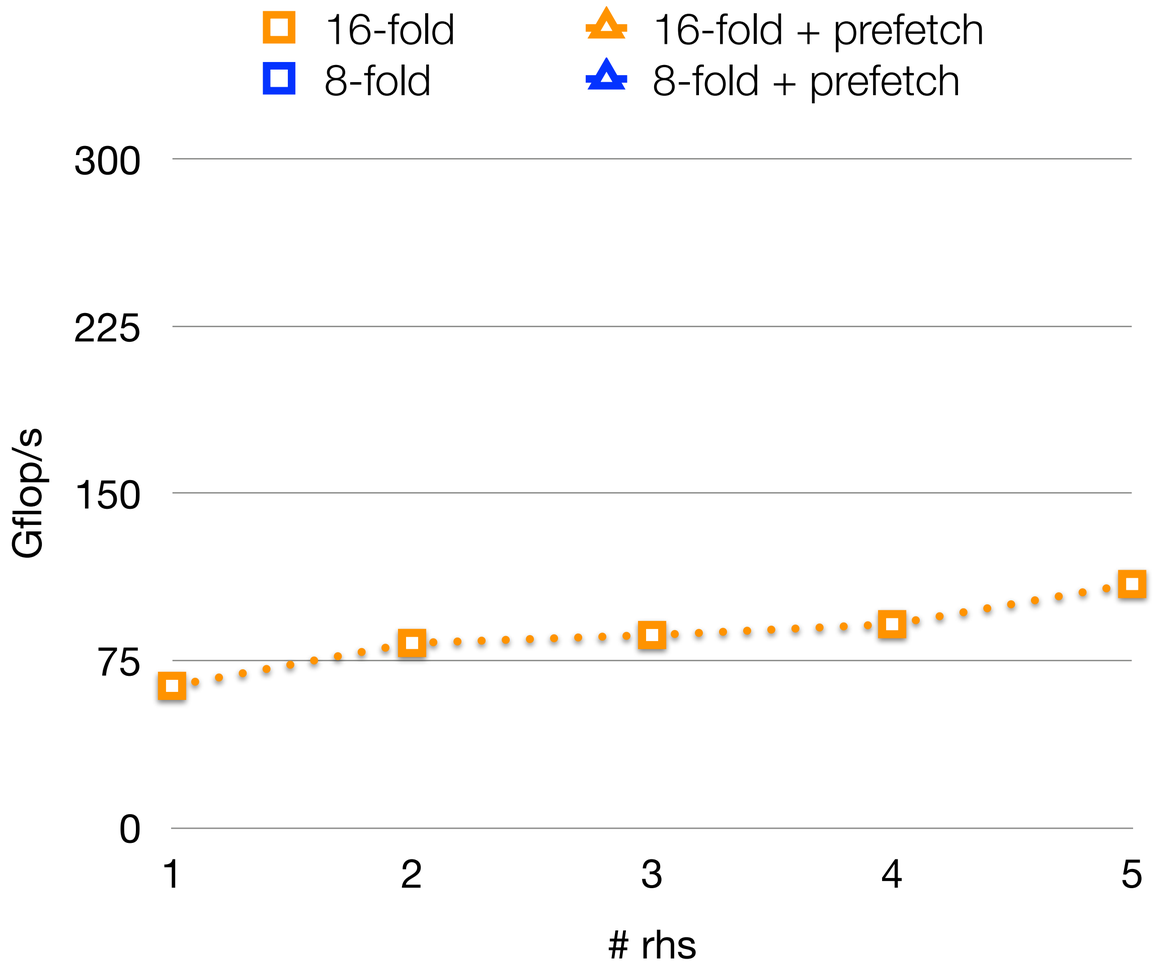
Some Details of the Phi Implementation

- effort lead by Patrick Steinbrecher (Universität Bielefeld → Brookhaven National Lab)
- single accelerator
- optimized for performance with multiple rhs
- parallelized using OpenMP
- vectorized using intrinsics:
  - fuse lattice sites into 512bit vectors
  - 16 sites with SoA-layout



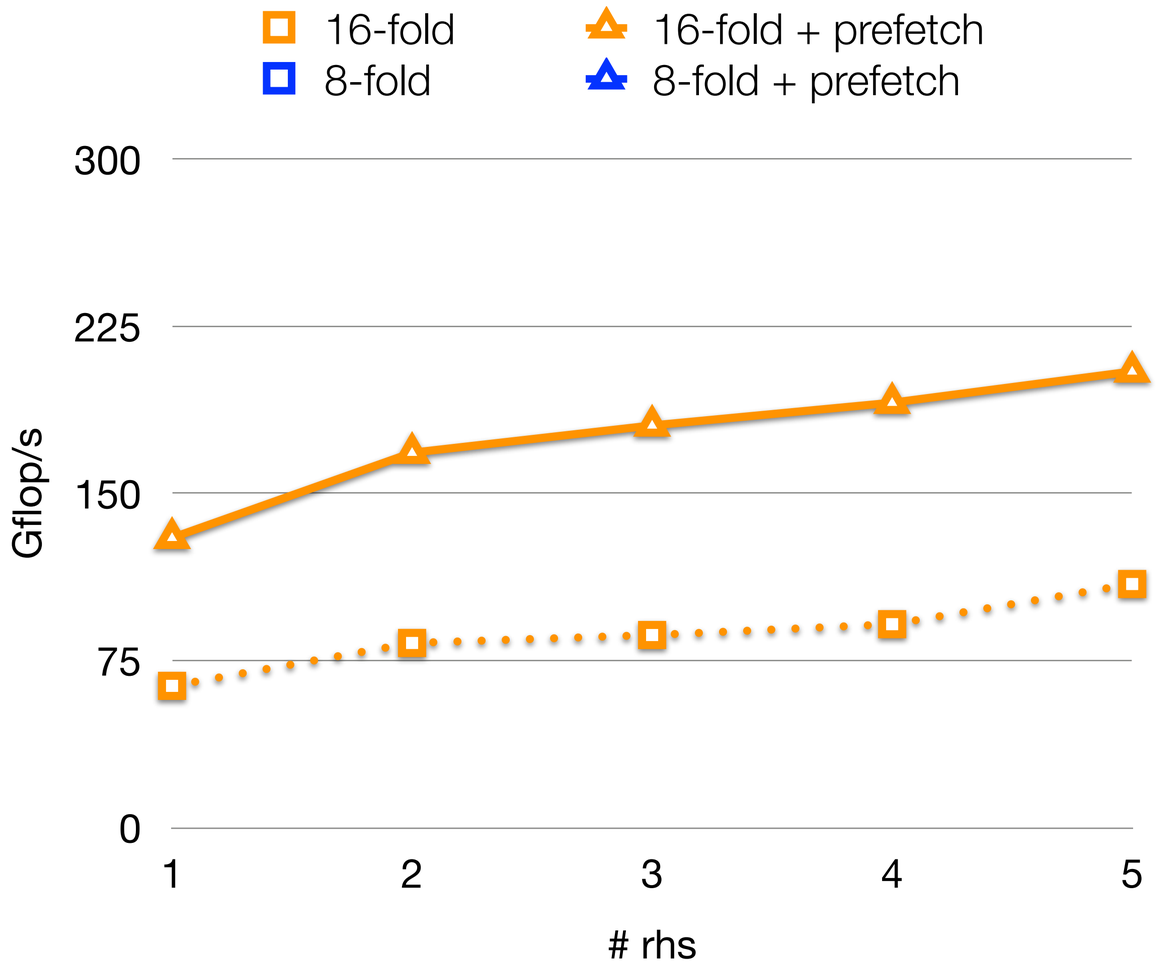
Impact of Memory Layout and Prefetch

- register pressure limits scaling with #rhs



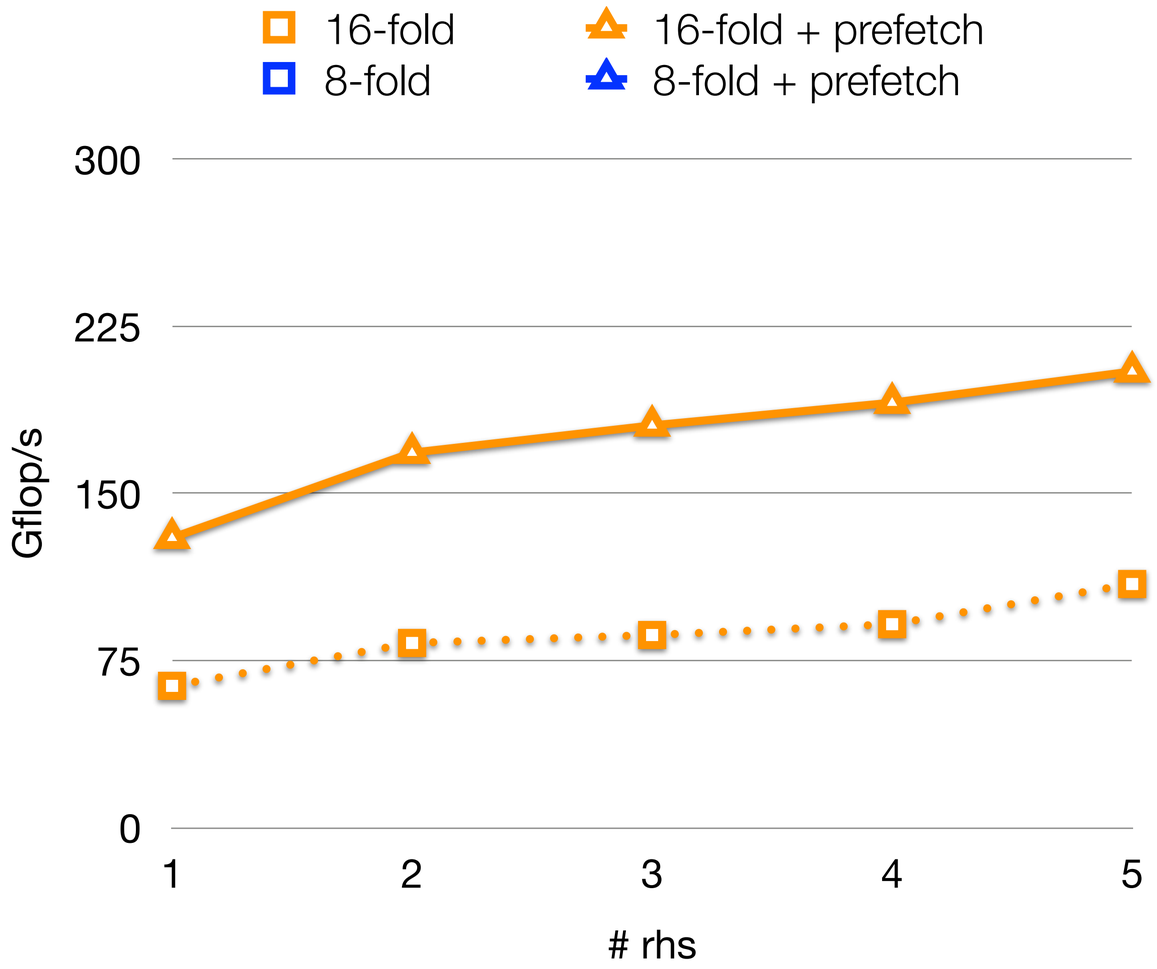
Impact of Memory Layout and Prefetch

- register pressure limits scaling with #rhs
- software prefetching improves by about 2x
  - hardware prefetching not effective for access pattern



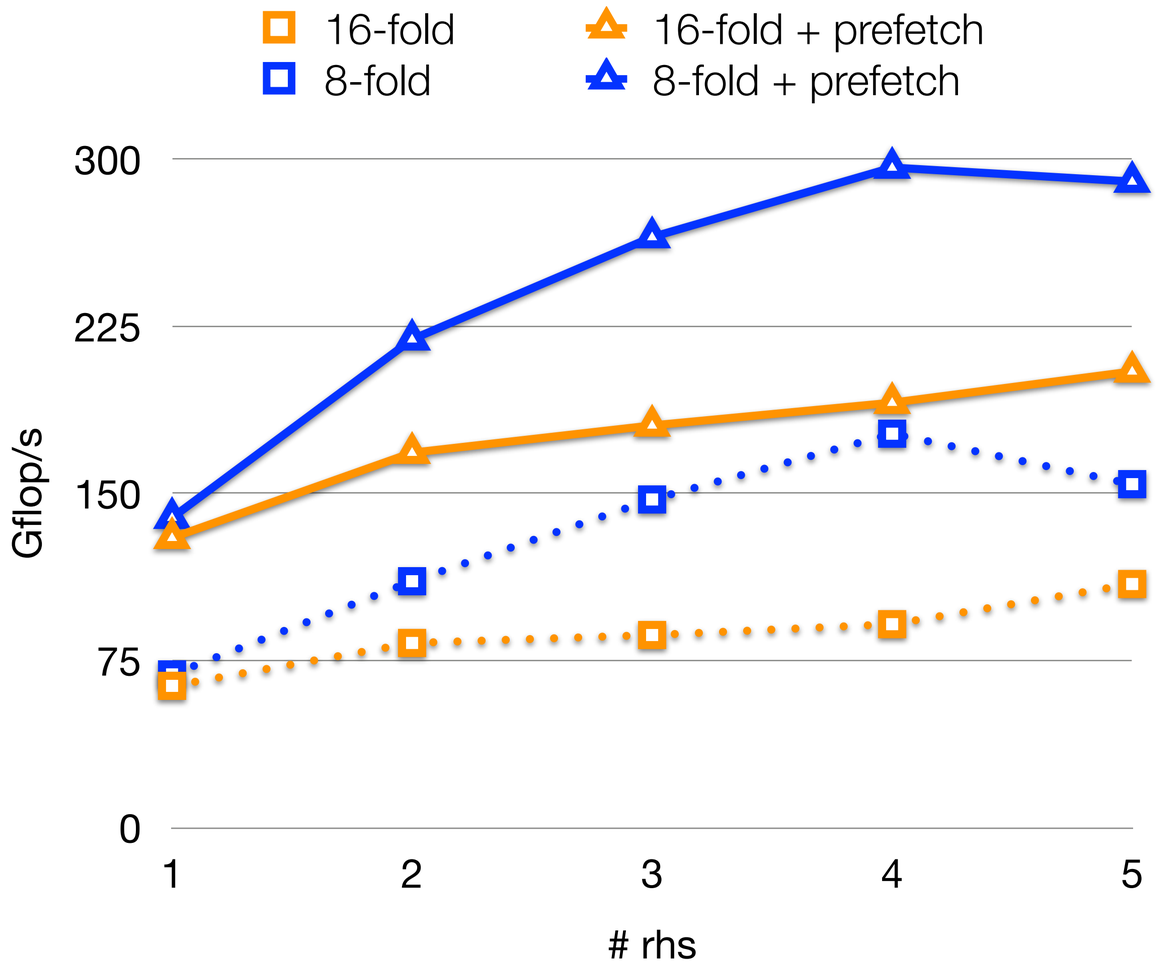
Impact of Memory Layout and Prefetch

- register pressure limits scaling with #rhs
- software prefetching improves by about 2x
  - hardware prefetching not effective for access pattern
- **8-fold site fusion**
  - reduces register pressure
  - harder to implement
  - small gain for 1 rhs



Impact of Memory Layout and Prefetch

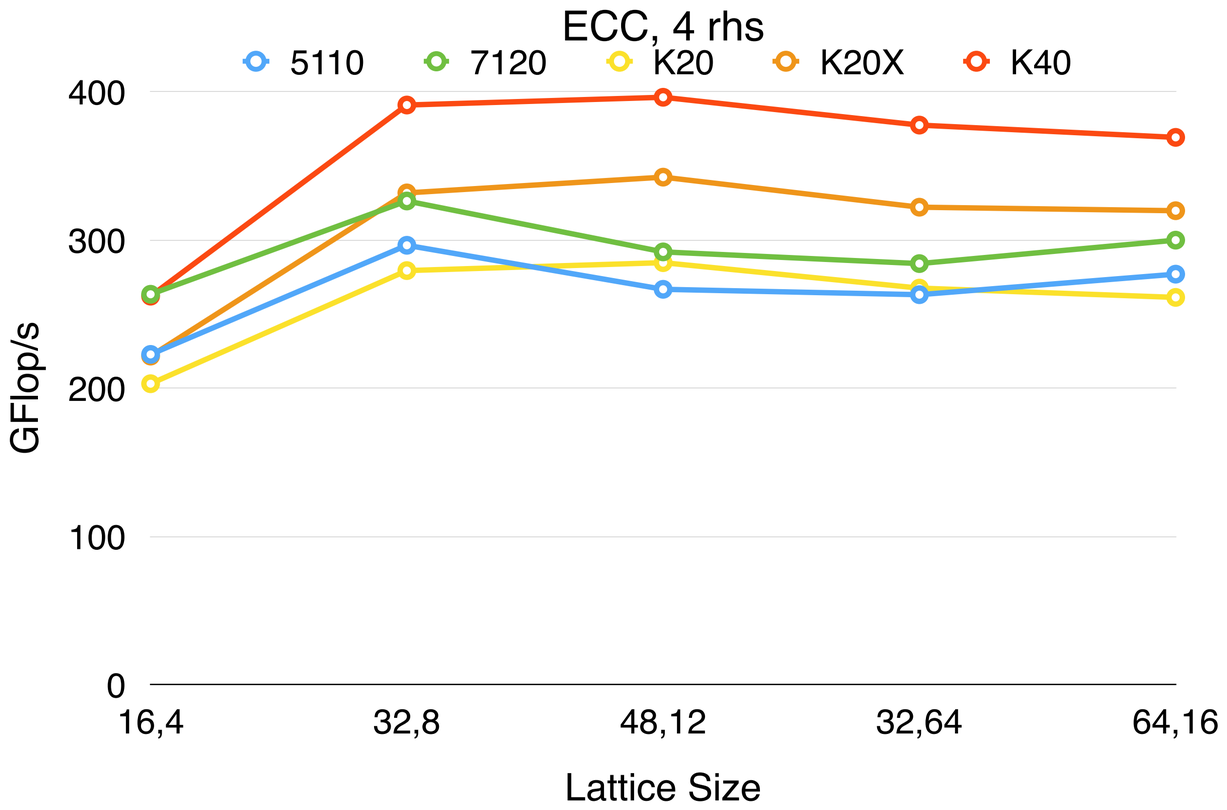
- register pressure limits scaling with #rhs
- software prefetching improves by about 2x
  - hardware prefetching not effective for access pattern
- **8-fold site fusion**
  - reduces register pressure
  - harder to implement
  - small gain for 1 rhs



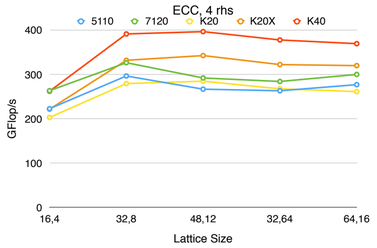
Let’s get ready to rumble

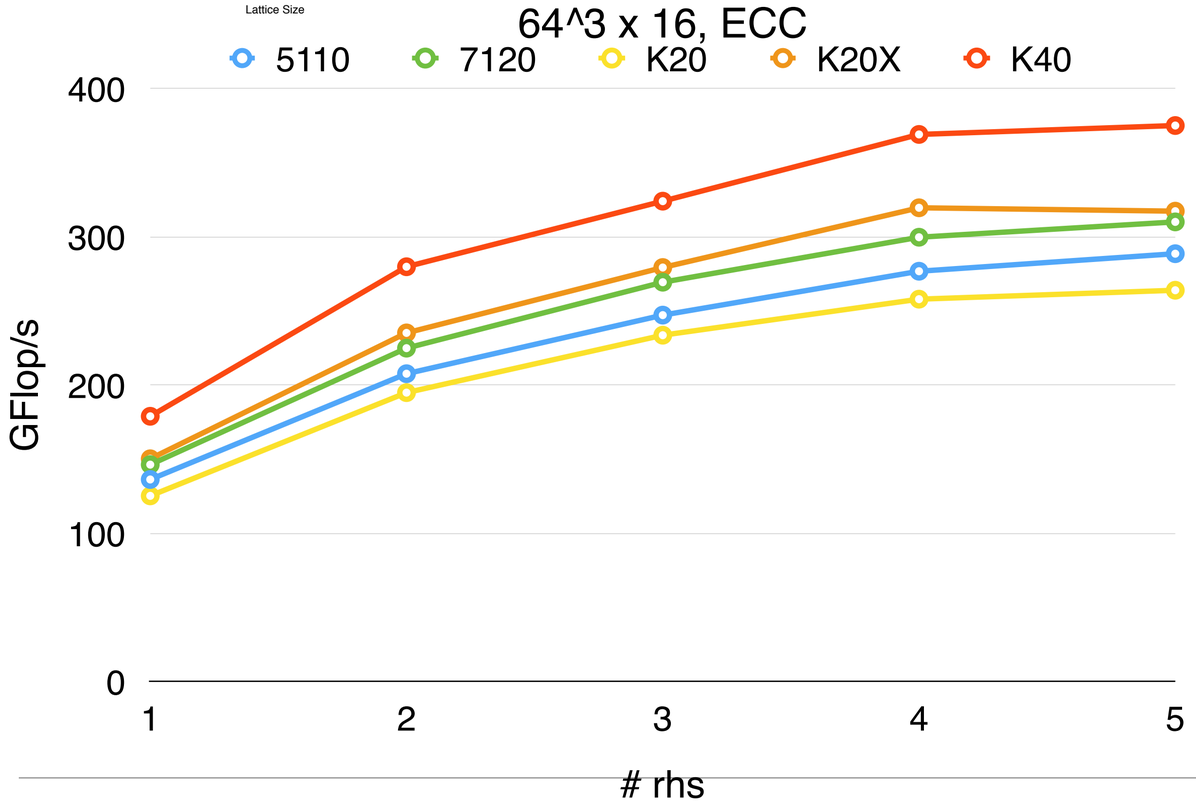
Results for the full conjugate gradient inverter on Xeon Phi and Tesla

Solver performance on KNL and Kepler

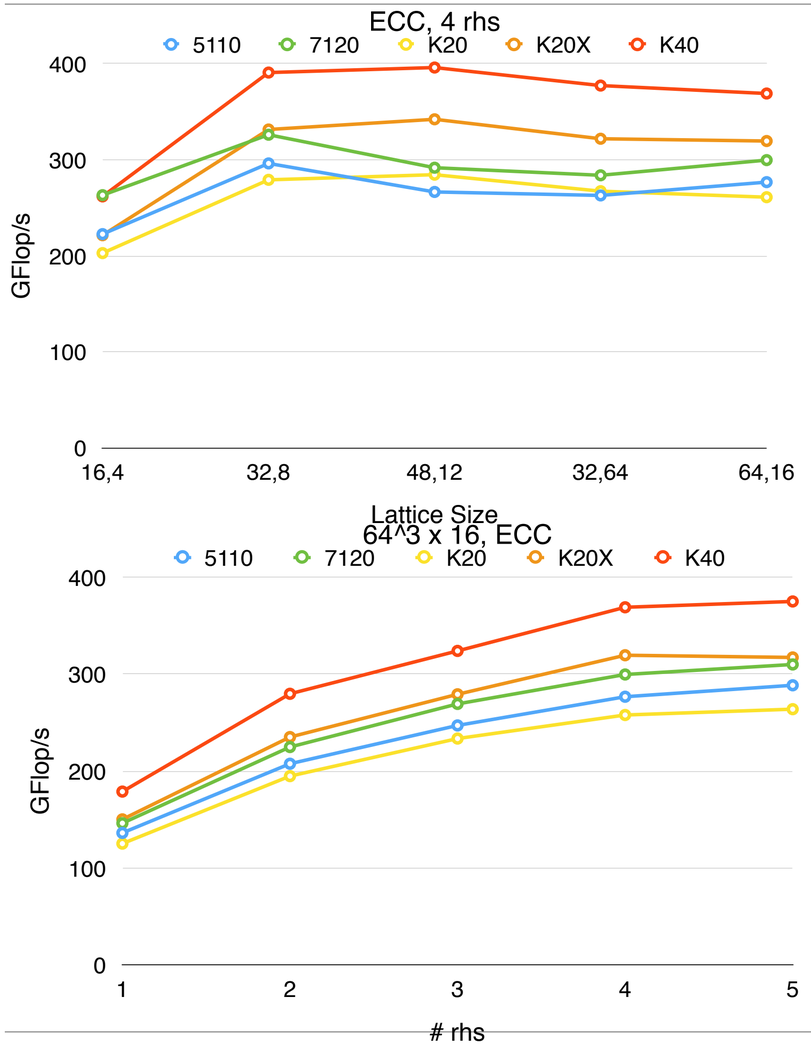


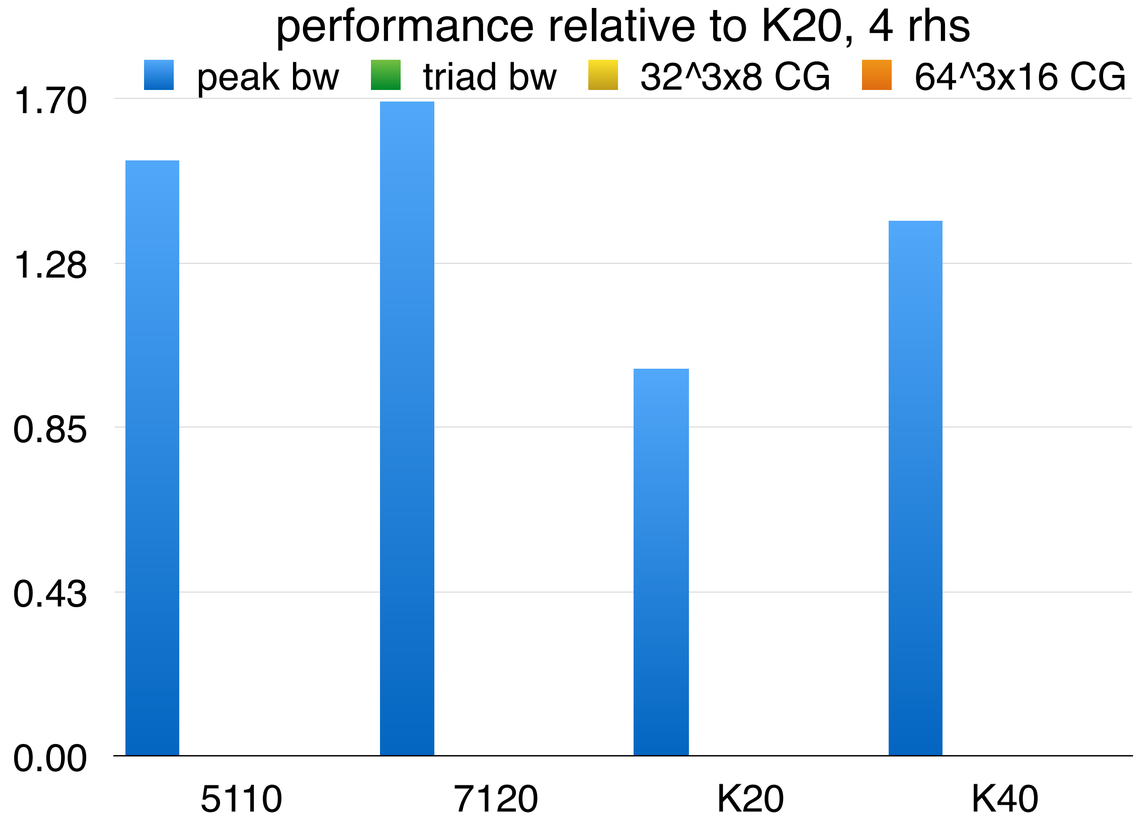
Solver performance on KNL and Kepler



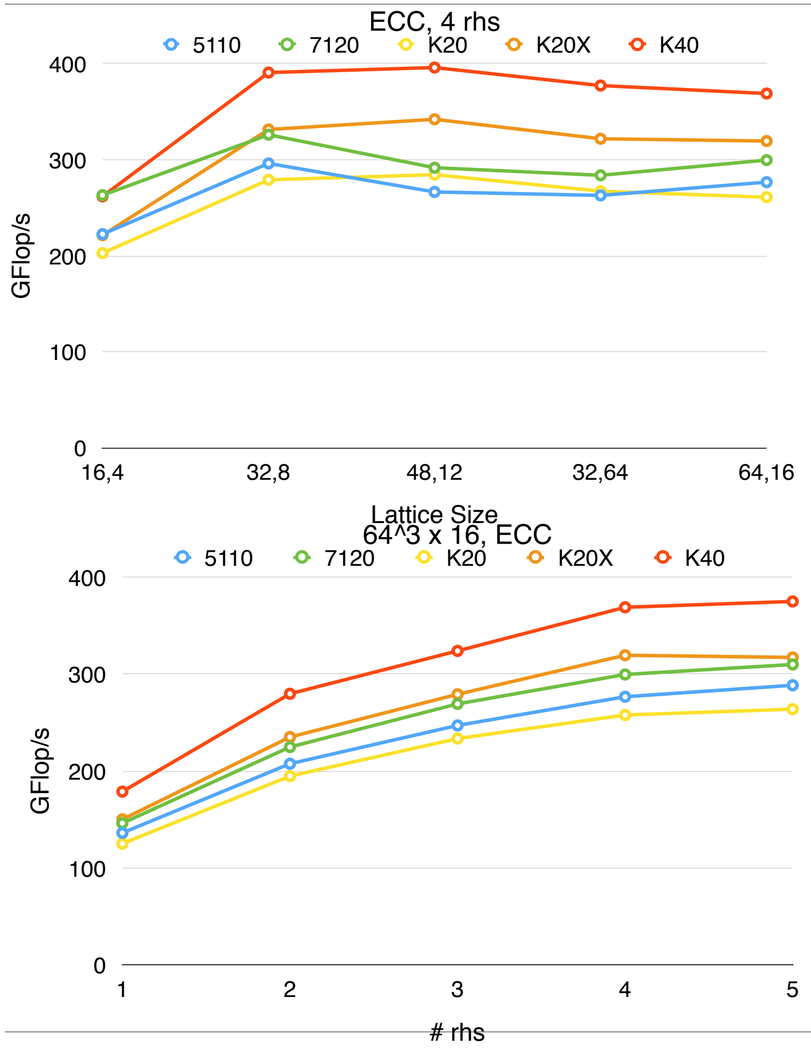


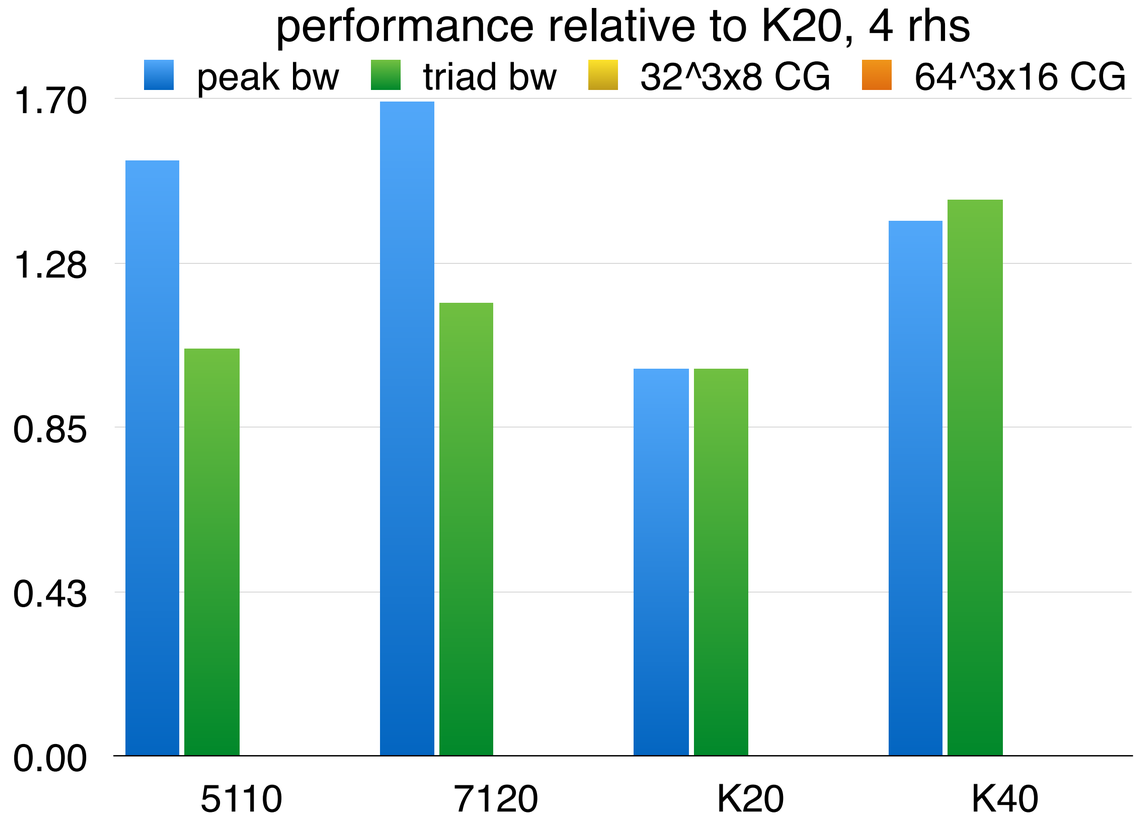
Solver performance on KNL and Kepler



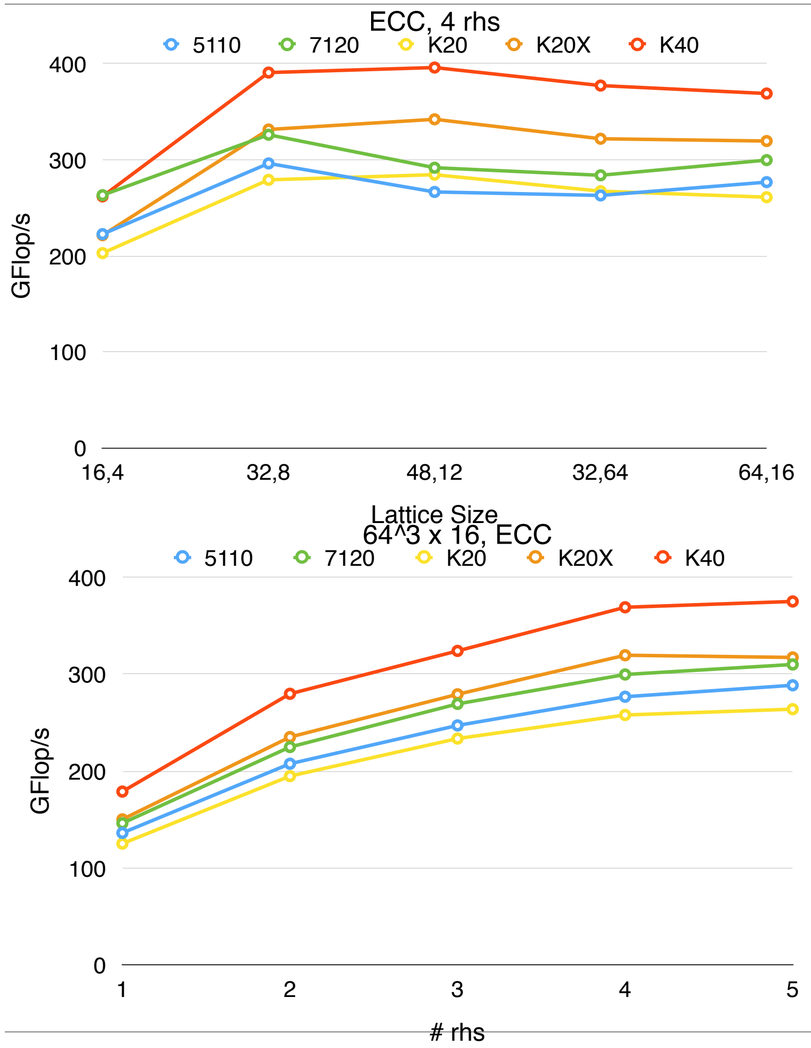


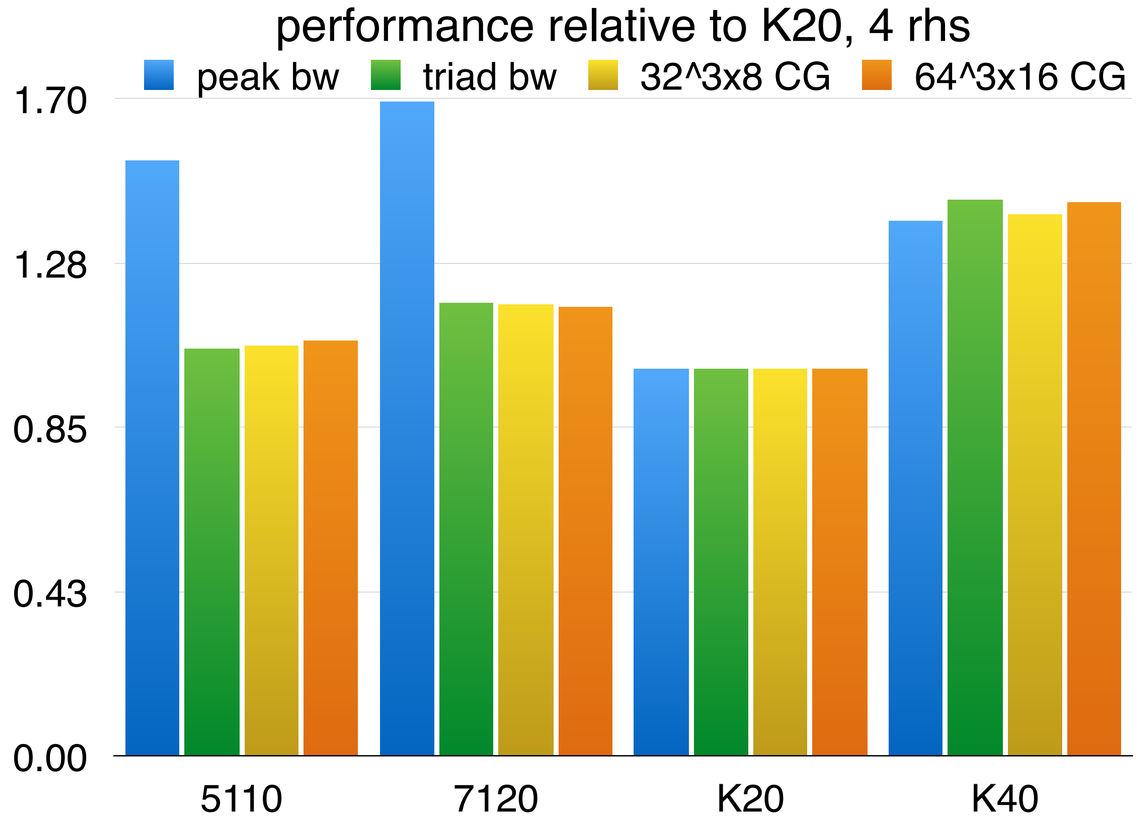
Solver performance on KNL and Kepler



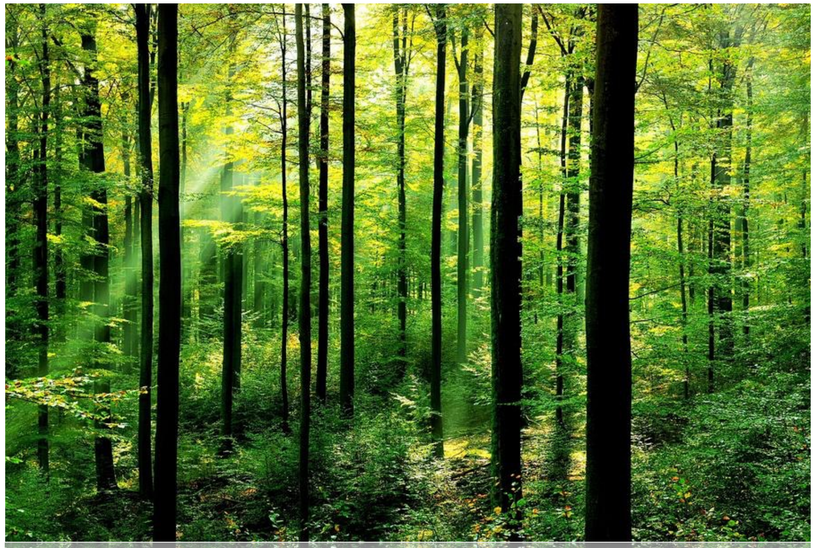


Solver performance on KNL and Kepler





Green or blue computing



How energy efficient are the two architectures?

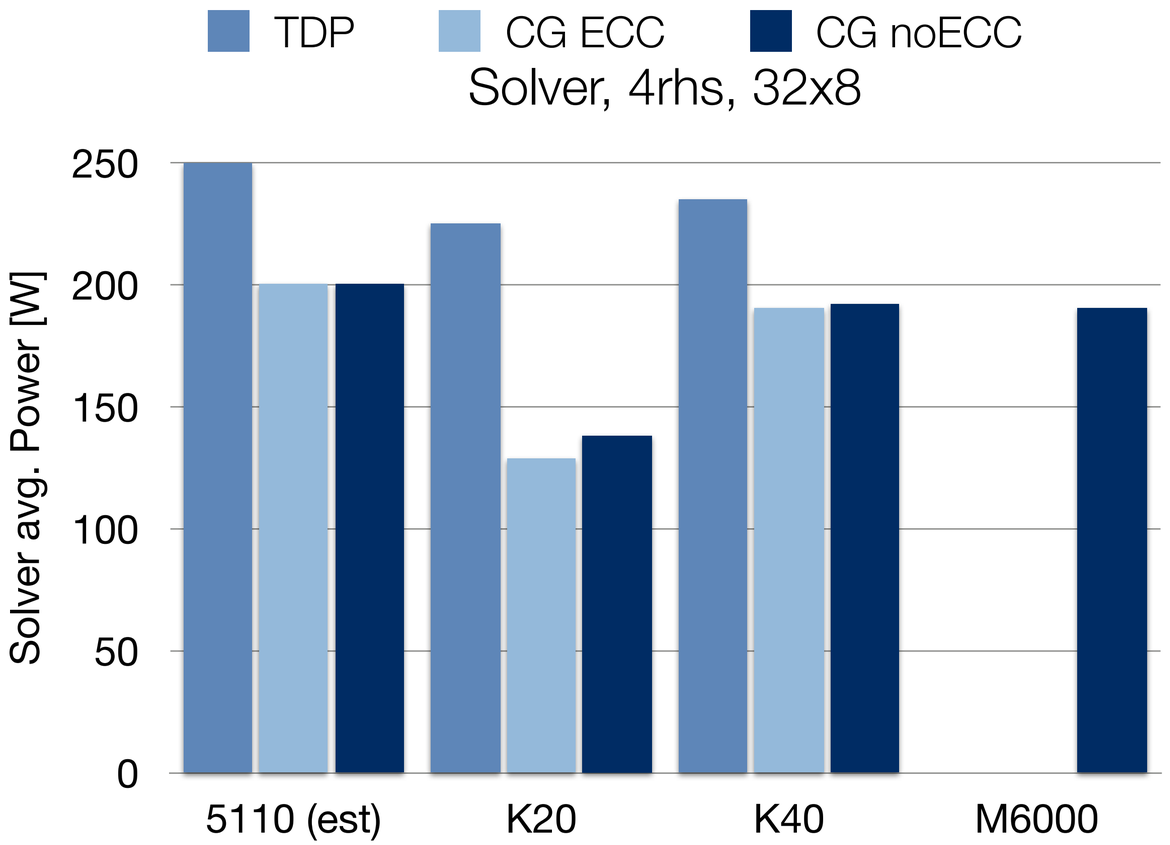
Oh, does anyone wonder about Maxwell in this respect?

Energy consumption

- bandwidth-bound applications are unlikely to hit TDP
- What is the relevant observable?
  - energy consumed by the node?
  - energy consumed by the accelerator?
  - include infrastructure (cooling, …) ?

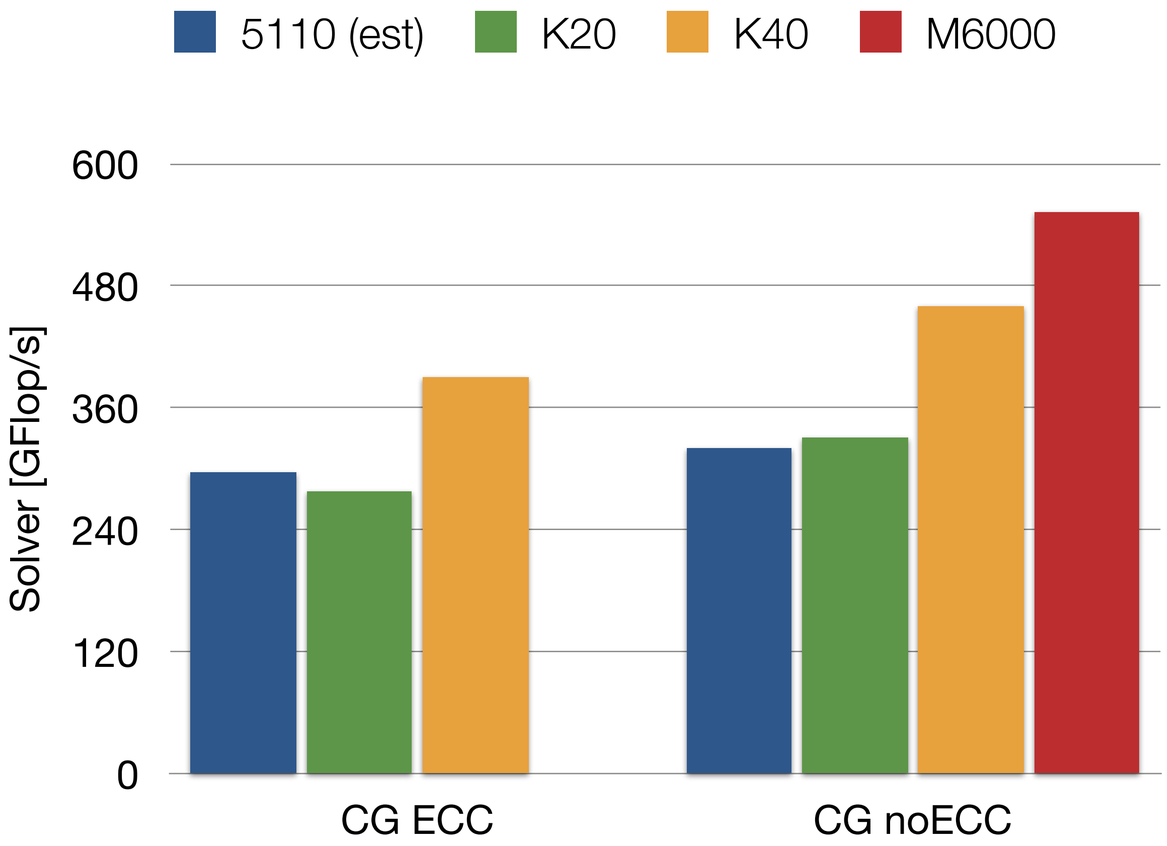
Energy consumption

- bandwidth-bound applications are unlikely to hit TDP
- What is the relevant observable?
  - energy consumed by the node?
  - energy consumed by the accelerator?
  - include infrastructure (cooling, …) ?
- Take what we can get
  - software reported power consumption (nvprof)
  - Xeon Phi is a bit more tricky: **estimate only**



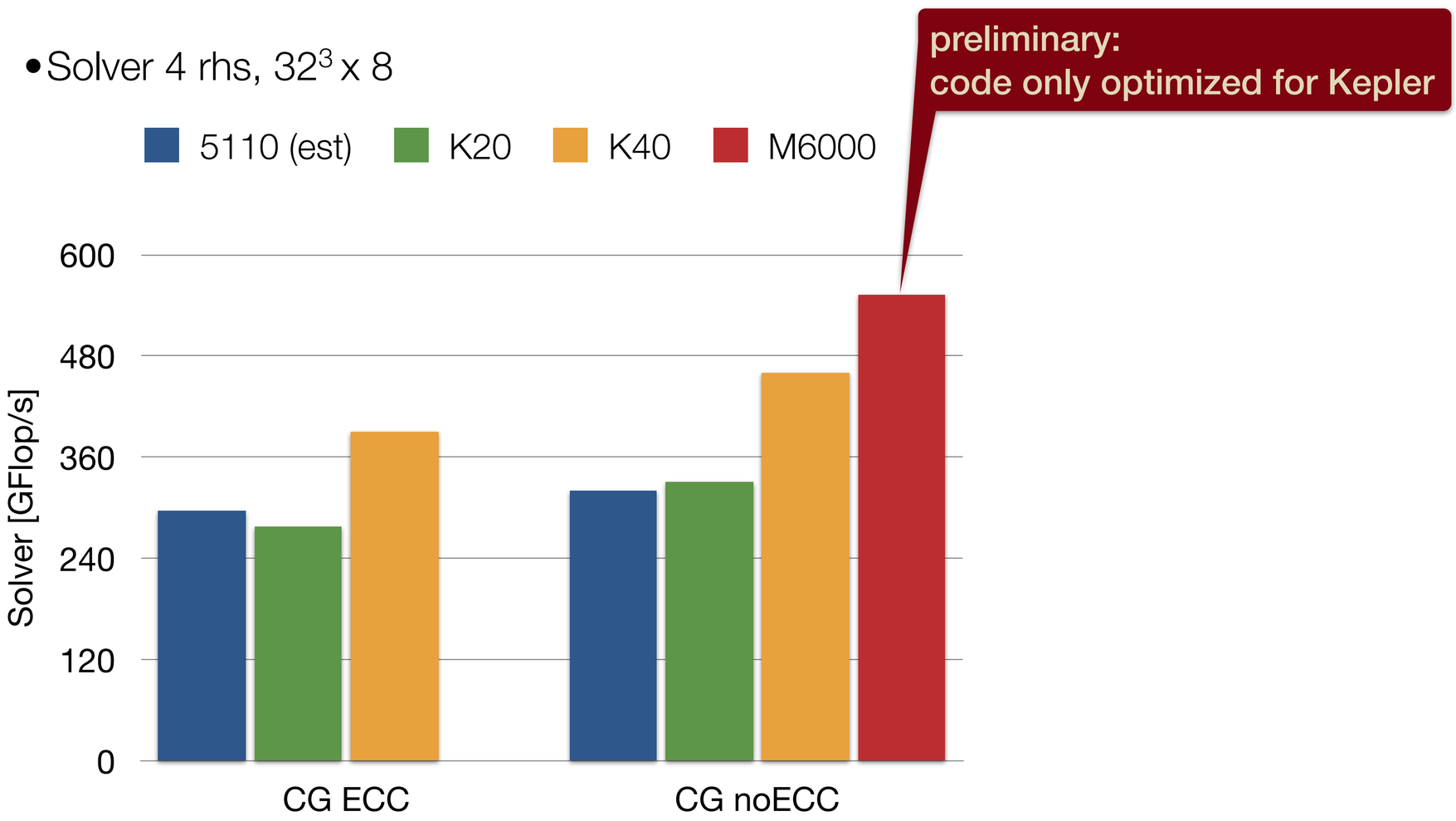
Performance per Watt

- Solver 4 rhs, 32^3 x 8



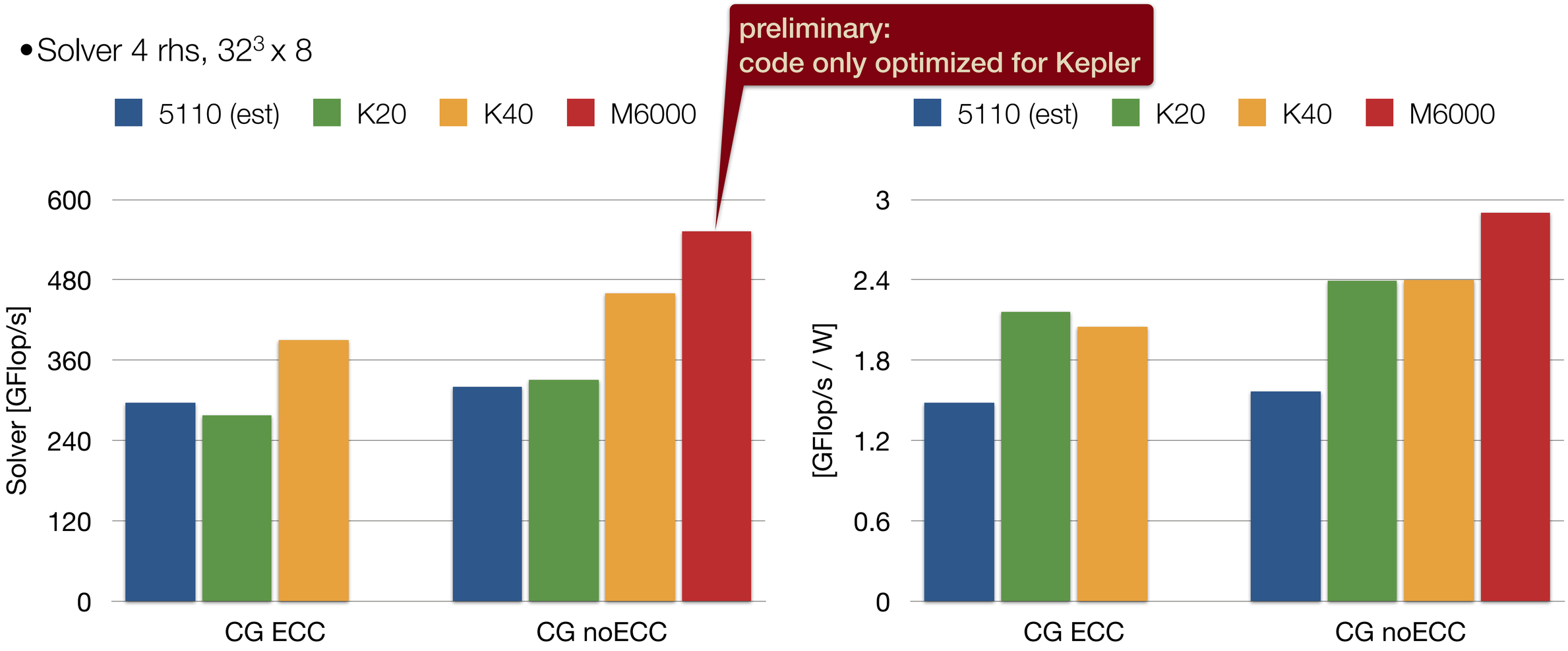
Performance per Watt

- Solver 4 rhs, 32^3 x 8



Performance per Watt

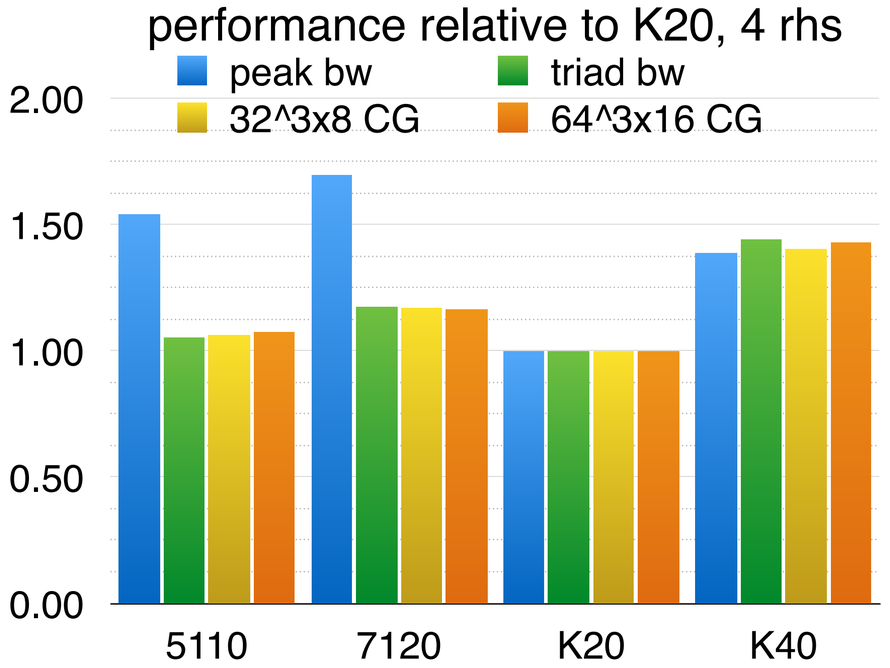
- Solver 4 rhs, $32^3 \times 8$



Finish

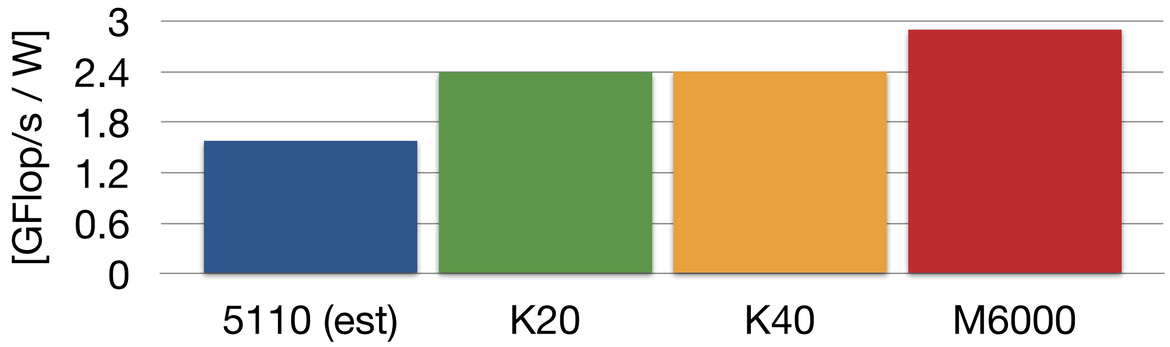
Summary

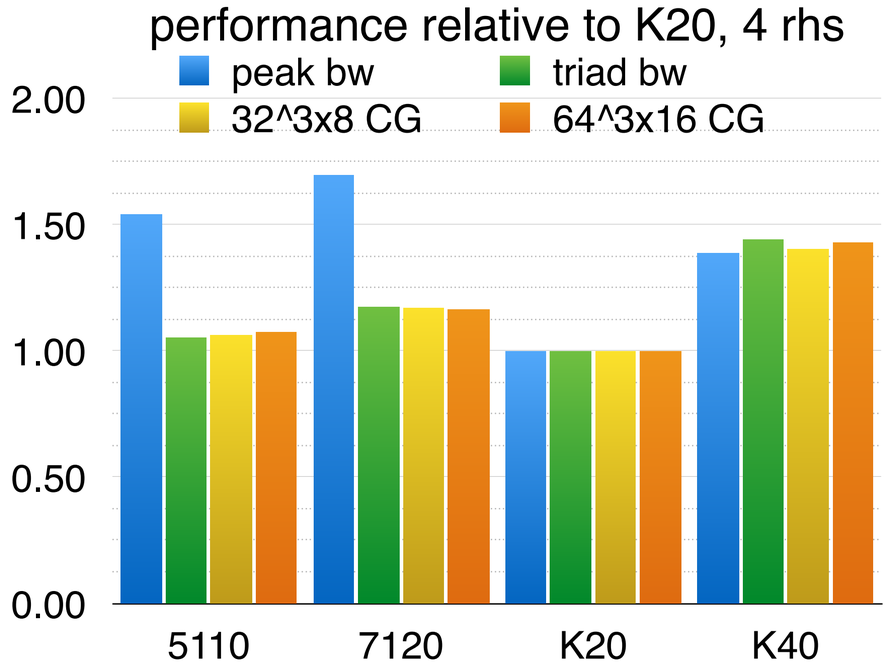
- Lattice QCD applications reflects triad bandwidth
  - equally well performing implementations for GPU / Phi
- multiple rhs achieve can easily speedup solver by 2.5
- Xeon Phi requires vectorization and software prefetches
- GPU uses texture cache
- Caching of vectors likely improved with multiple rhs



Summary

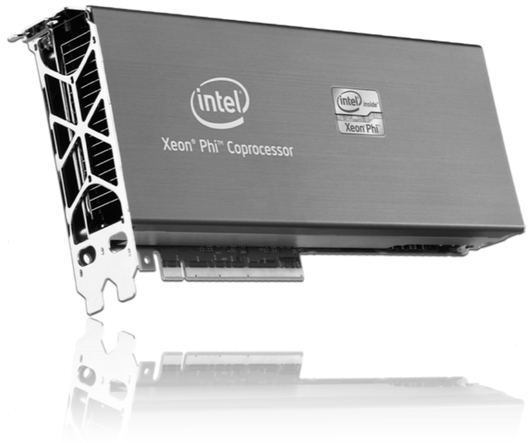
- Lattice QCD applications reflects triad bandwidth
  - equally well performing implementations for GPU / Phi
- multiple rhs achieve can easily speedup solver by 2.5
- Xeon Phi requires vectorization and software prefetches
- GPU uses texture cache
- Caching of vectors likely improved with multiple rhs





- GK110 about 1.5 times more efficient than KNL
- Maxwell promises another factor 1.5
- multiple rhs about twice as energy efficient

GPU vs Xeon Phi: Performance of Bandwidth Bound Applications with a Lattice QCD Case Study



Contact:
mathwagn@indiana.edu
http://linked.in/mathwagn
@mathwagn

Thanks to:
Jeongnim Kim (Intel)
Mike Clark (Nvidia)

Collaborators:
P. Steinbrecher (Bielefeld U → Brookhaven National Lab)
C. Schmidt (Bielefeld U)
O. Kaczmarek (Bielefeld U)

References:
arXiv:1411.4439 [physics.comp-ph]
arXiv:1409.1510 [cs.DC]