Implicit and Explicit Optimizations for Stencil Computations

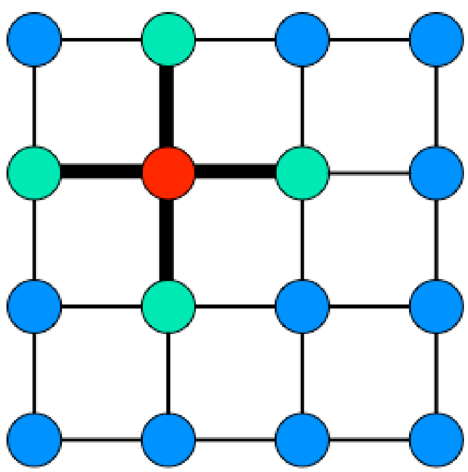
By Shoaib Kamil[1,2], Kaushik Datta[1], Samuel Williams[1,2], Leonid Oliker[2], John Shalf[2] and Katherine A. Yelick[1,2]

[1]BeBOP Project, U.C. Berkeley
[2]Lawrence Berkeley National Laboratory
October 22, 2006

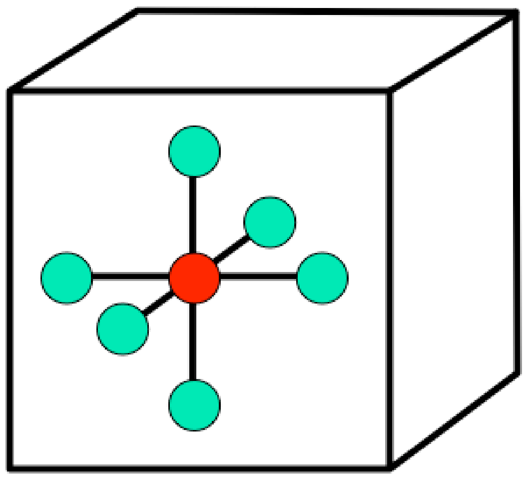
http://bebop.cs.berkeley.edu
kdatta@eecs.berkeley.edu

What are stencil codes?

- For a given point, a *stencil* is a pre-determined set of nearest neighbors (possibly including itself)
- A *stencil code* updates every point in a regular grid with a weighted subset of its neighbors ("applying a stencil")



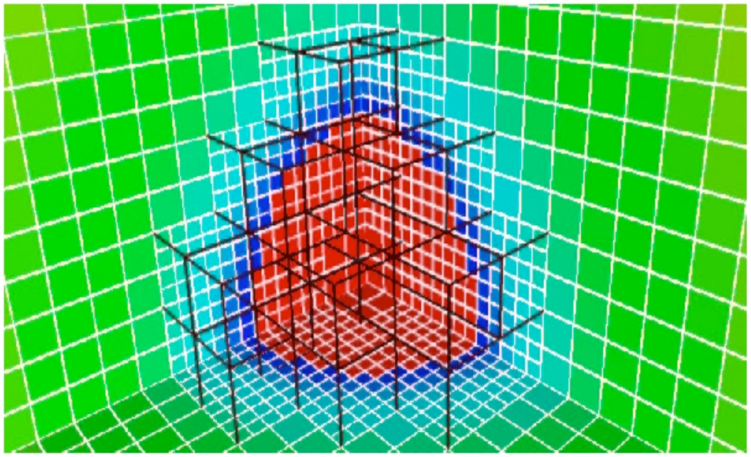
2D Stencil



3D Stencil

Stencil Applications

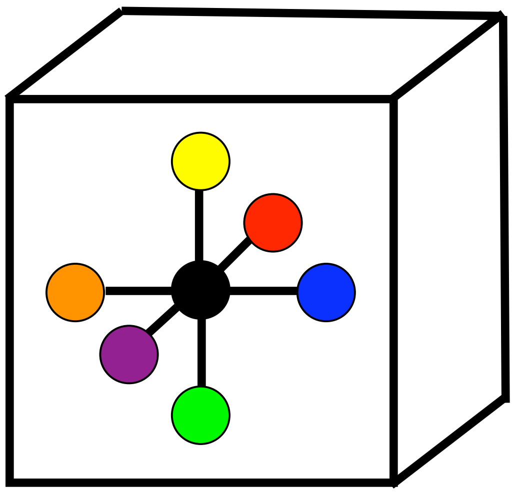
- Stencils are critical to many scientific applications:
  - Diffusion, Electromagnetics, Computational Fluid Dynamics
  - Both explicit and implicit iterative methods (e.g. Multigrid)
  - Both uniform and adaptive block-structured meshes
- Many type of stencils
  - 1D, 2D, 3D meshes
  - Number of neighbors (5-pt, 7-pt, 9-pt, 27-pt,…)
  - Gauss-Seidel (update in place) vs Jacobi iterations (2 meshes)



- Our study focuses on 3D, 7-point, Jacobi iteration

Naïve Stencil Pseudocode (One iteration)

```
void stencil3d(double A[], double B[], int nx, int ny, int nz) {
  for all grid indices in x-dim {
    for all grid indices in y-dim {
      for all grid indices in z-dim {
        B[center] = S0* A[center] +
                    S1*(A[top] + A[bottom] +
                        A[left] + A[right] +
                        A[front] + A[back]);
      }
    }
  }
}
```



Potential Optimizations

- Performance is limited by memory bandwidth and latency
  - Re-use is limited to the number of neighbors in a stencil
  - For large meshes (e.g., 512^3), cache blocking helps
  - For smaller meshes, stencil time is roughly the time to read the mesh once from main memory
  - Tradeoff of blocking: reduces cache misses (bandwidth), but increases prefetch misses (latency)
  - See previous paper for details [Kamil et al, MSP '05]
- We look at merging across iterations to improve reuse
  - Three techniques with varying level of control
- We vary architecture types
  - Significant work (not shown) on low level optimizations

Optimization Strategies

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
- Two hardware techniques
  - Fast memory (*cache*) is managed by hardware
  - Fast memory (*local store*) is managed by application software

Software \ Hardware	Cache (Implicit)	Local Store (Explicit)
Conscious (Explicit)	Cache Conscious	Cache Conscious on Cell
Oblivious (Implicit)	Cache Oblivious	N/A

If hardware forces control, software cannot be oblivious

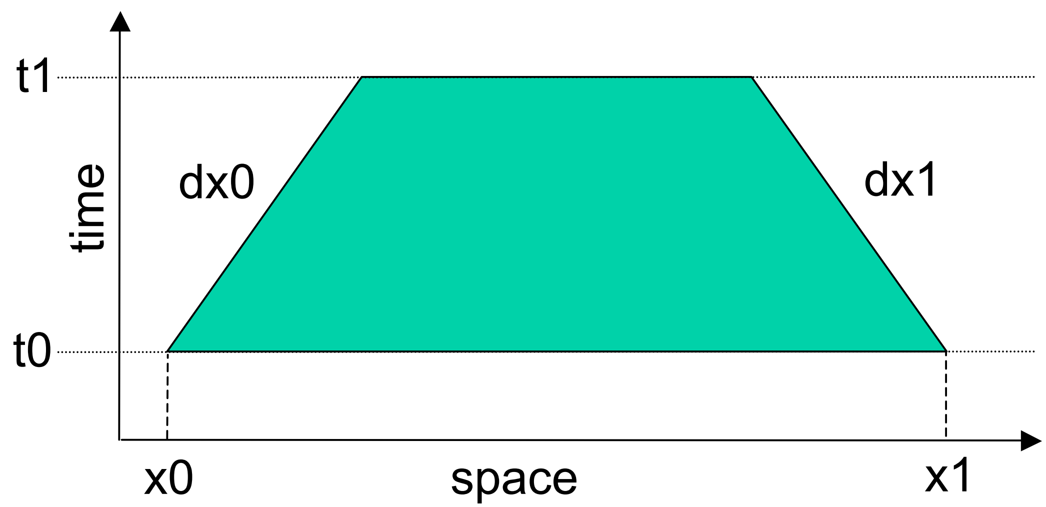
Opt. Strategy #1: Cache Oblivious

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
    - Elegant Solution
    - No explicit block size
    - No need to tune block size
  - *Cache conscious* has an explicit block size
- Two hardware techniques
  - Cache managed by hw
    - Less programmer effort
  - Local store managed by sw

Software \ Hardware	Cache (Implicit)	Local Store (Explicit)
Conscious (Explicit)	Cache Conscious	Cache Conscious on Cell
Oblivious (Implicit)	Cache Oblivious	N/A

Cache Oblivious Algorithm

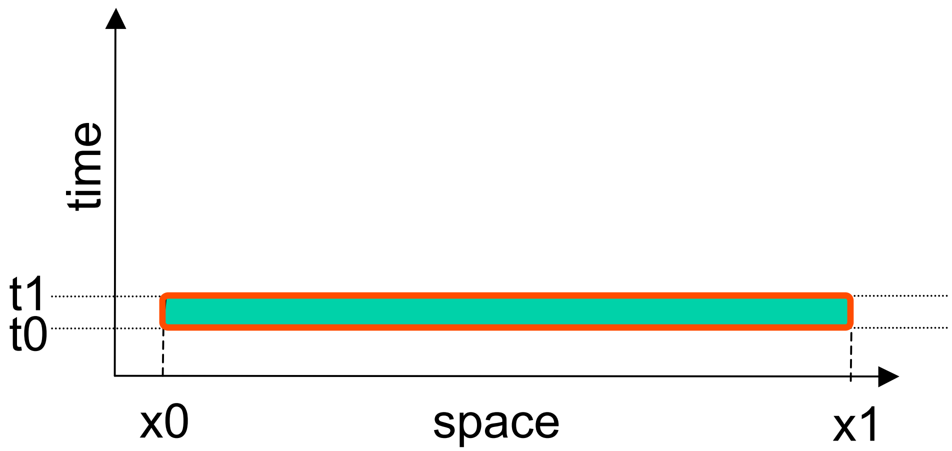
- By Matteo Frigo et al
- Recursive algorithm consists of *space cuts*, *time cuts*, and a base case
- Operates on well-defined trapezoid (x0, dx0, x1, dx1, t0, t1):



- Trapezoid for 1D problem; our experiments are for 3D (shrinking cube)

Cache Oblivious Algorithm - Base Case

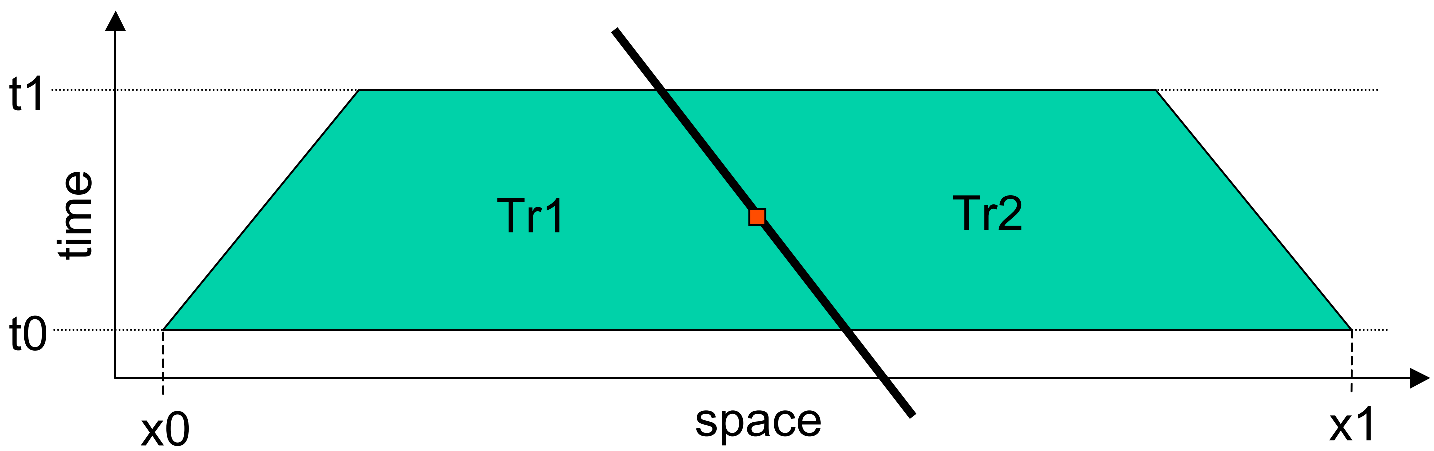
- If the height=1, then we have a line of points (x0:x1, t0):



- At this point, we stop the recursion and perform the stencil on this set of points
- Order does not matter since there are no inter-dependencies

Cache Oblivious Algorithm - Space Cut

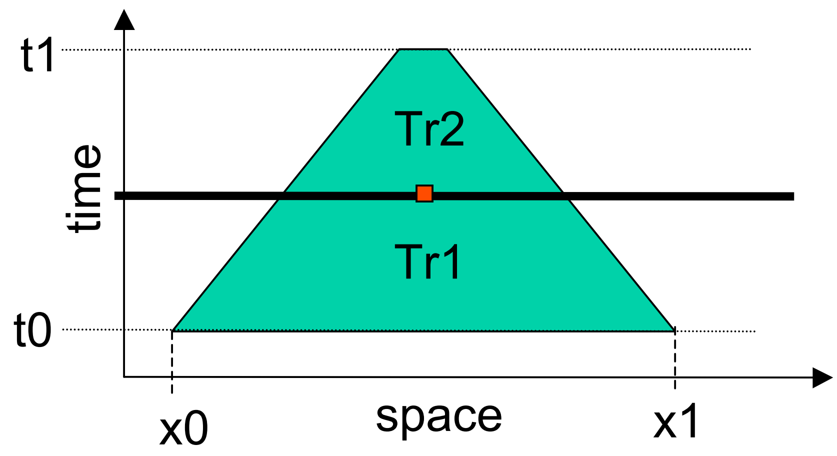
- If trapezoid width >= 2*height, cut with slope=-1 through the center:



- Since no point in Tr1 depends on Tr2, execute Tr1 first and then Tr2
- In multiple dimensions, we try space cuts in each dimension before proceeding

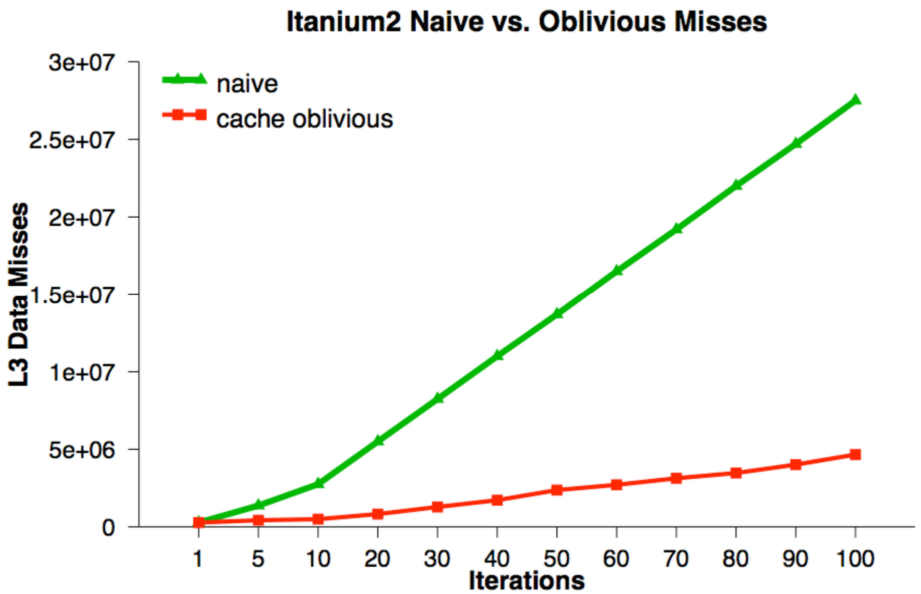
Cache Oblivious Algorithm - Time Cut

- Otherwise, cut the trapezoid in half in the time dimension:

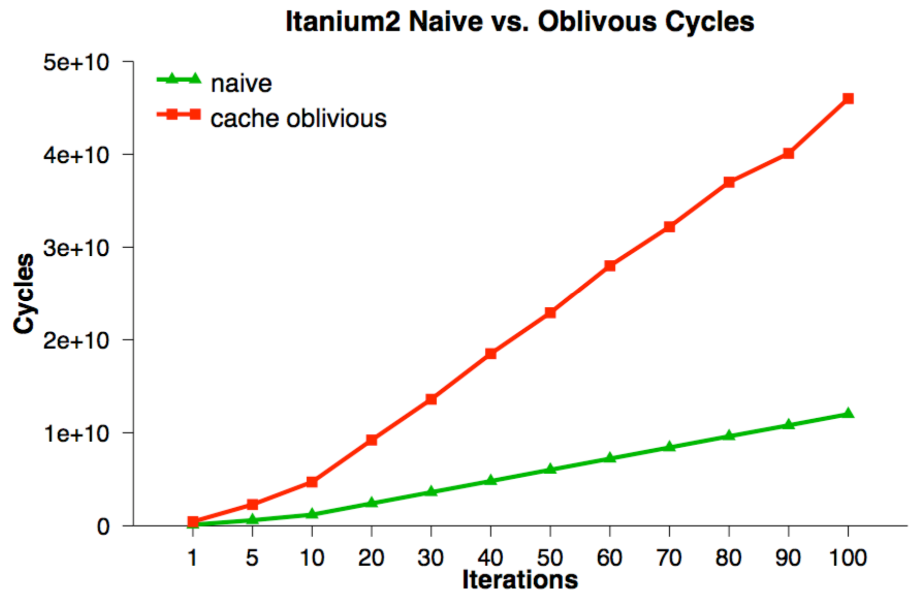


- Again, since no point in Tr1 depends on Tr2, execute Tr1 first and then Tr2

Poor Itanium 2 Cache Oblivious Performance



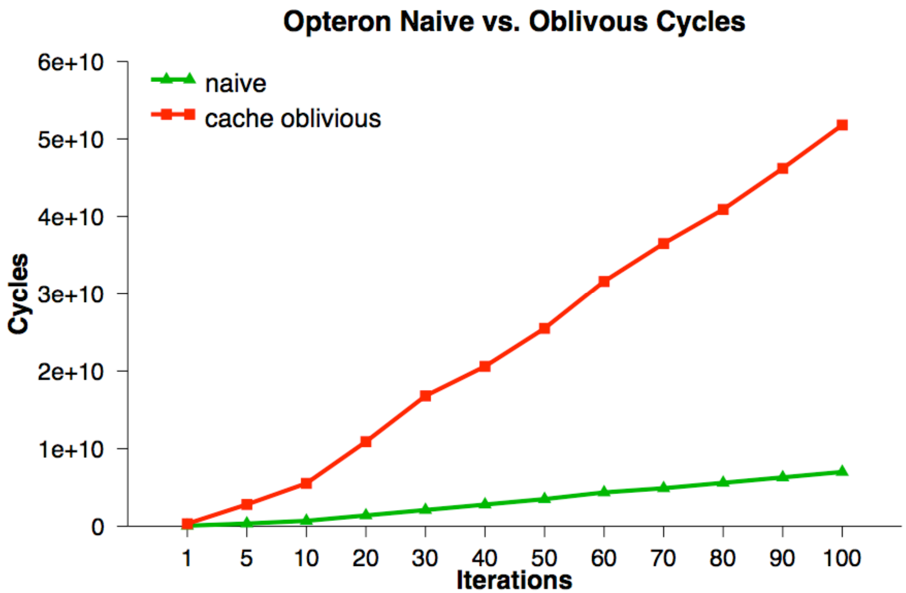
L3 Cache Miss Comparison



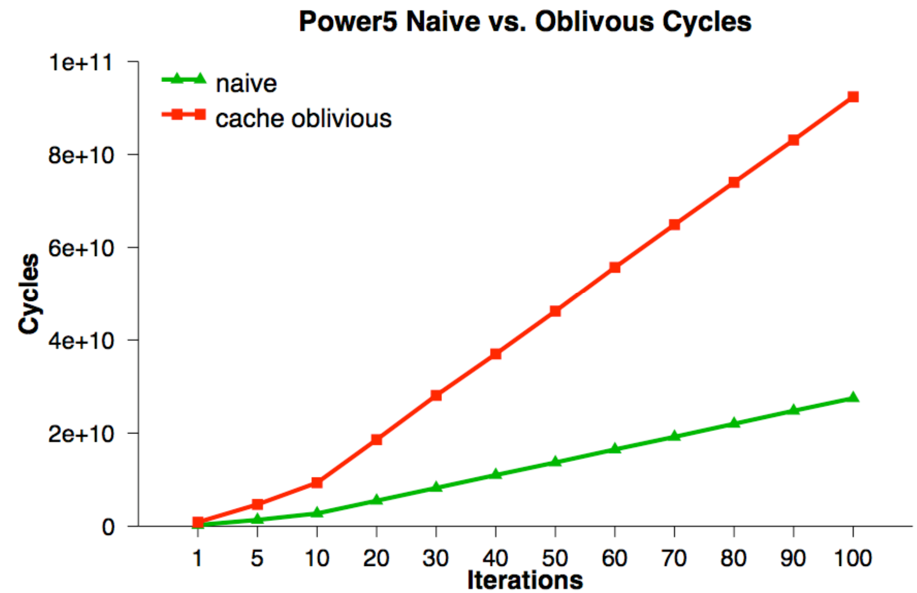
Cycle Comparison

- Fewer cache misses BUT longer running time

Poor Cache Oblivious Performance



Opteron Cycle Comparison



Power5 Cycle Comparison

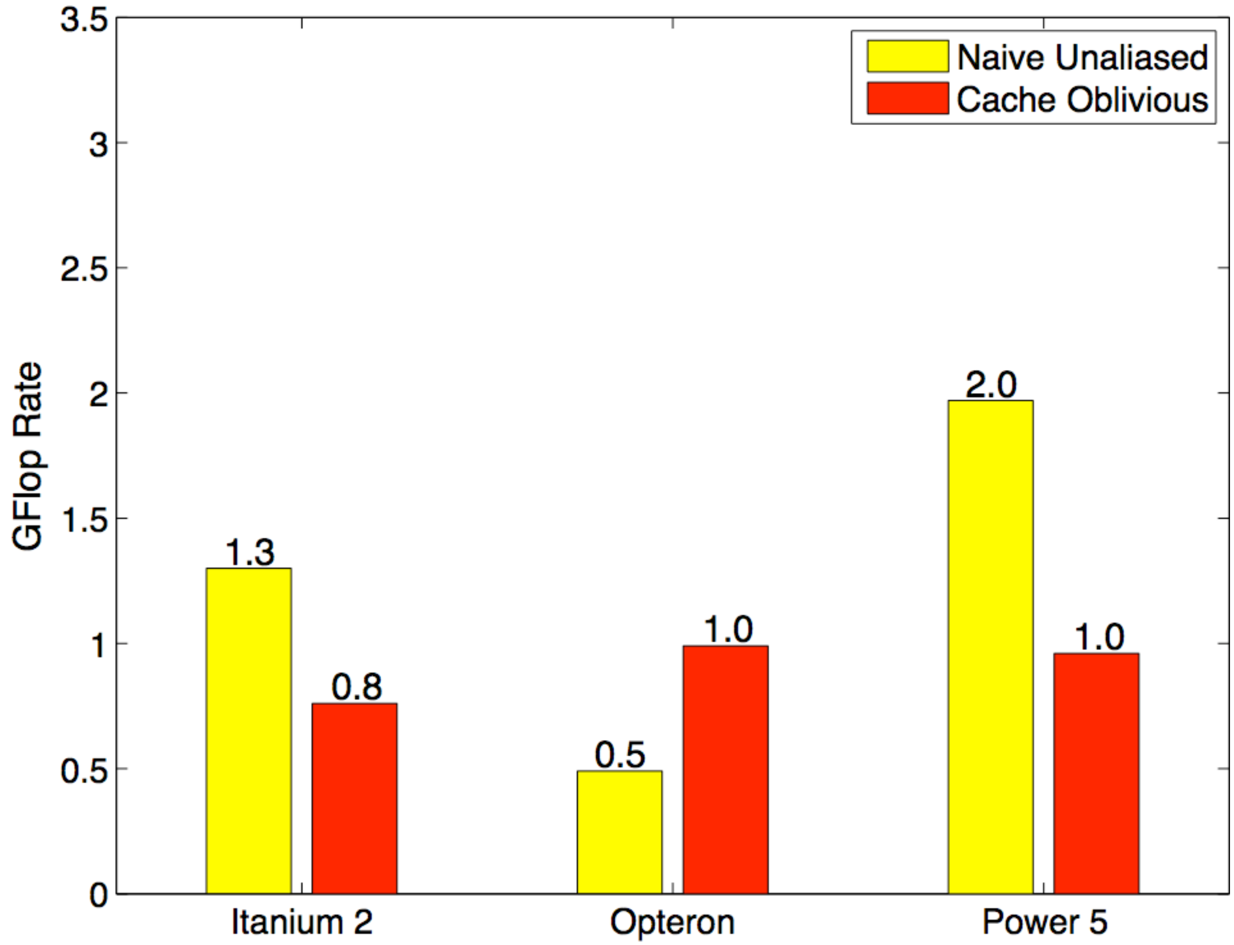
- Much slower on Opteron and Power5 too

Improving Cache Oblivious Performance

- Fewer cache misses did NOT translate to better performance:

Problem	Solution
Extra function calls	Inlined kernel
Poor prefetch behavior	No cuts in unit-stride dimension
Recursion stack overhead	Maintain explicit stack
Modulo Operator	Pre-computed lookup array
Recursion even after block fits in cache	Early cut off of recursion

Cache Oblivious Performance



- Only Opteron shows any benefit

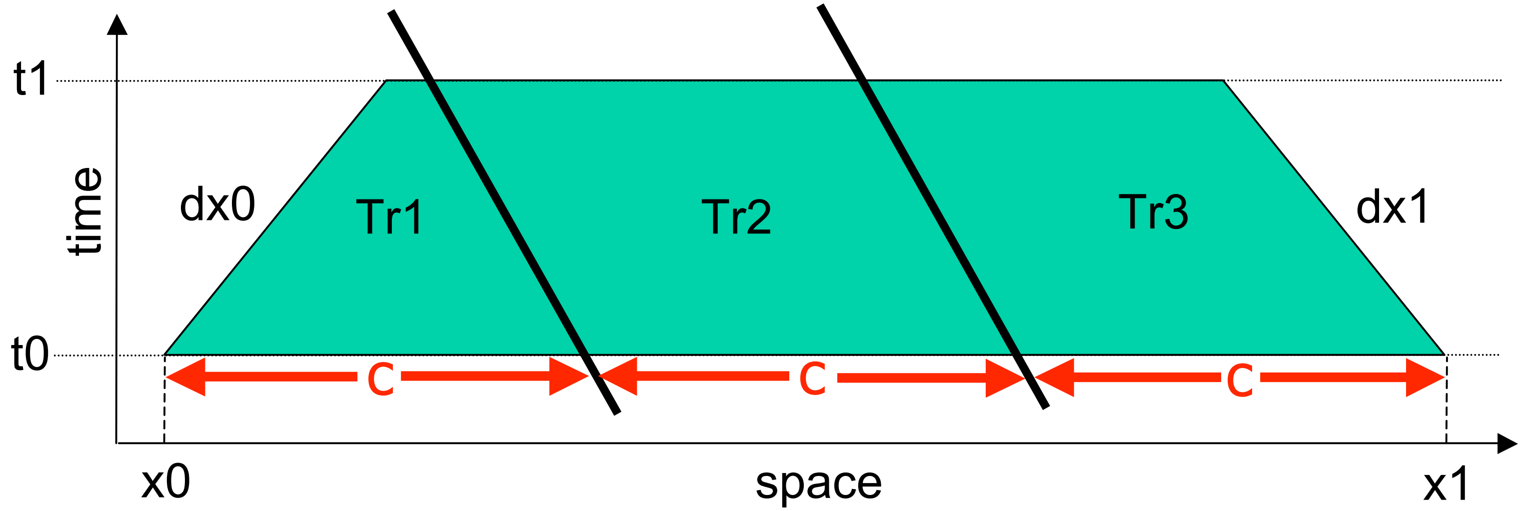
Opt. Strategy #2: Cache Conscious

- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
    - Easier to visualize
    - Tunable block size
    - No recursion stack overhead
- Two hardware techniques
  - Cache managed by hw
    - Less programmer effort
  - Local store managed by sw

Software \ Hardware	Cache (Implicit)	Local Store (Explicit)
Conscious (Explicit)	Cache Conscious	Cache Conscious on Cell
Oblivious (Implicit)	Cache Oblivious	N/A

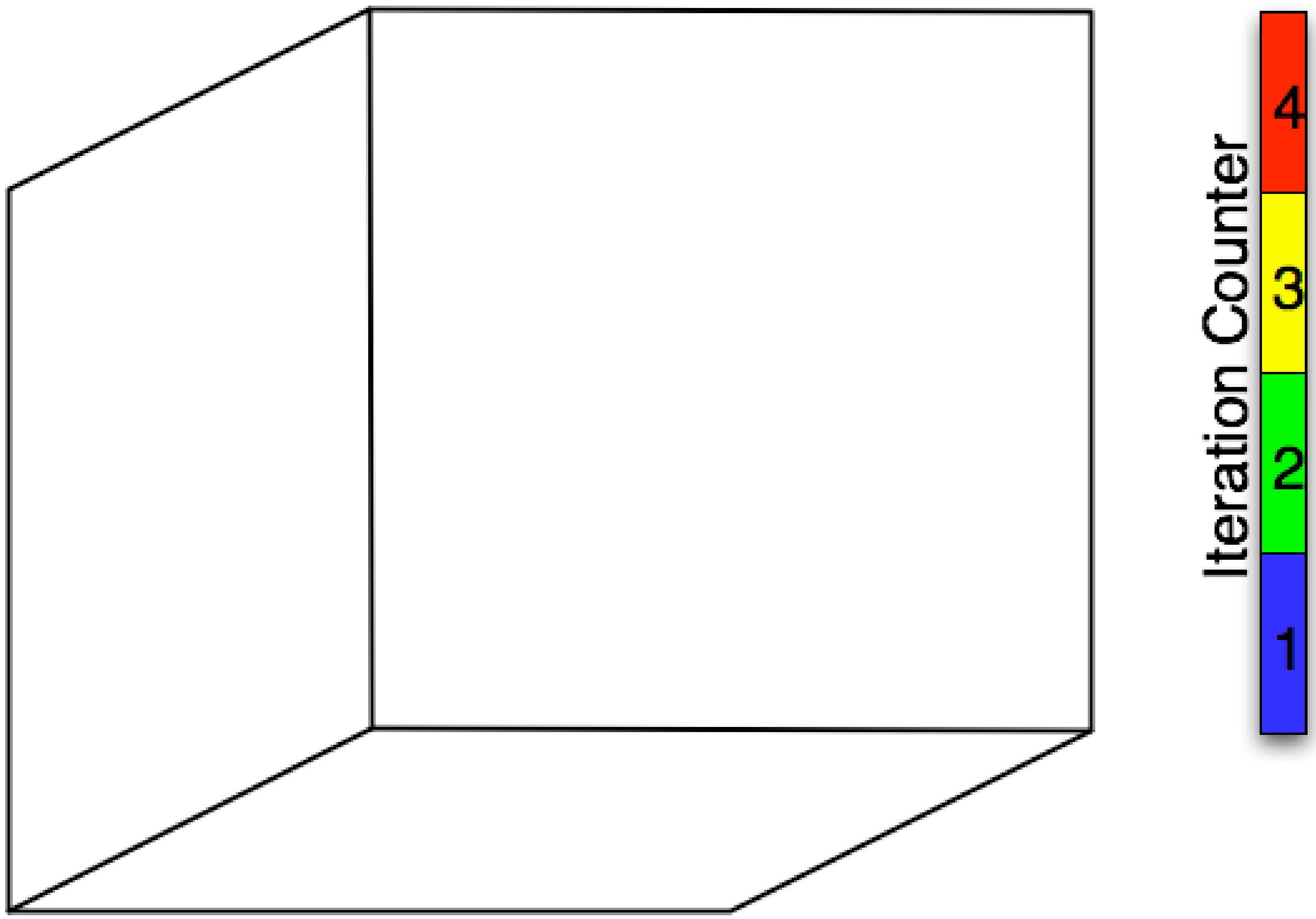
Cache Conscious Algorithm

- Like the cache oblivious algorithm, we have space cuts
- However, cache conscious is NOT recursive and *explicitly* requires cache block dimension c as a parameter

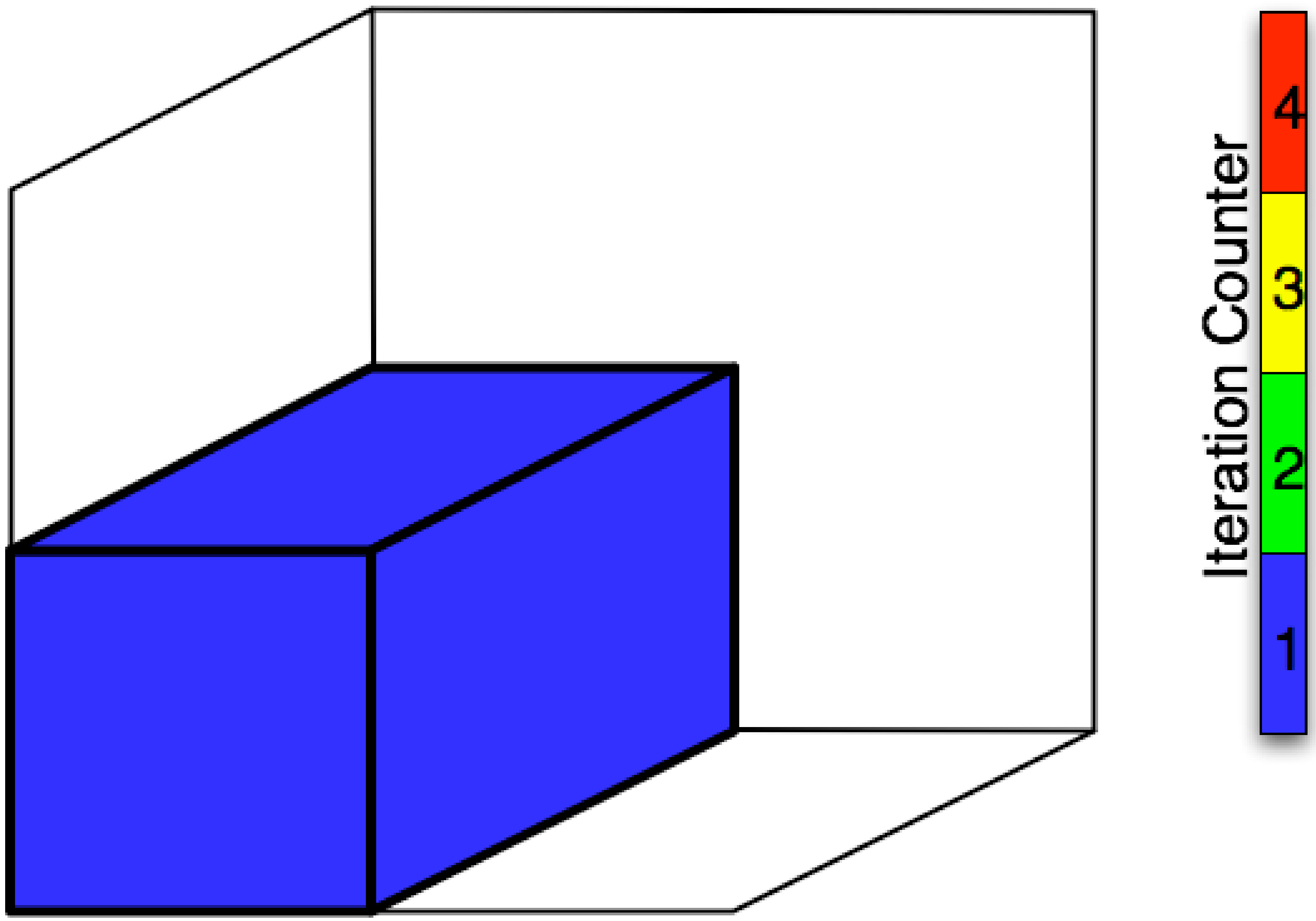


- Again, trapezoid for a 1D problem above

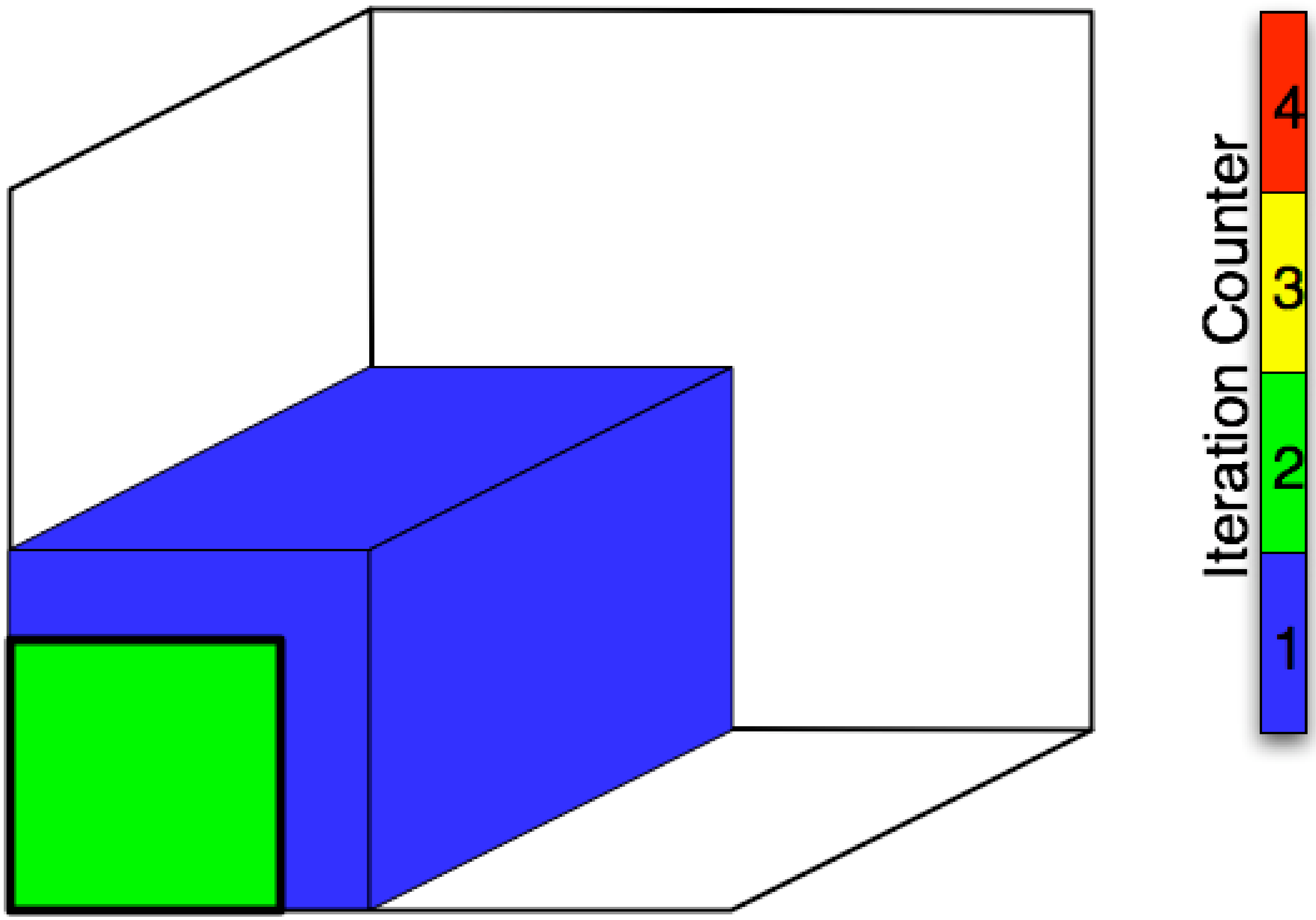
Cache Conscious - 3D Animation



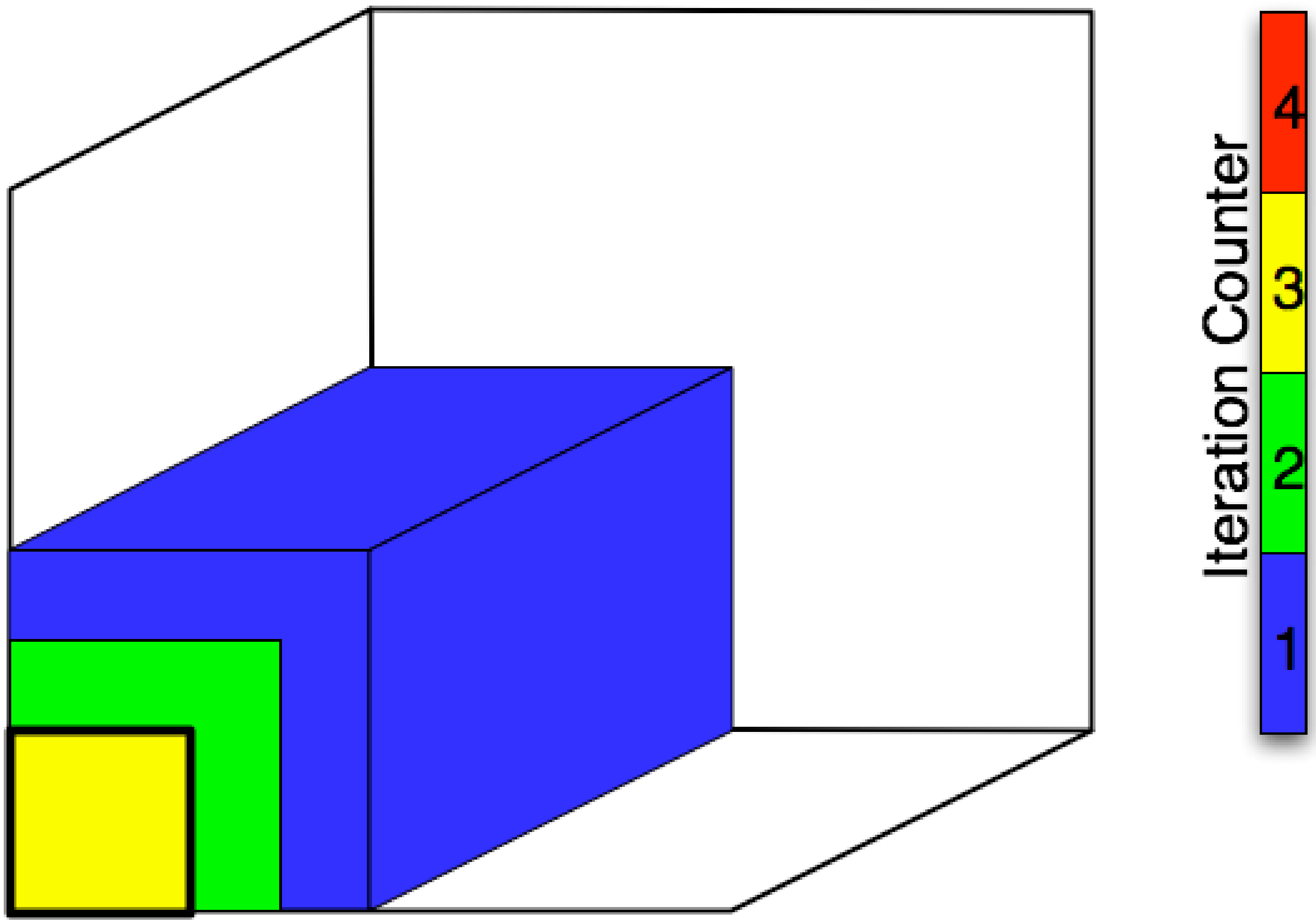
Cache Conscious - 3D Animation



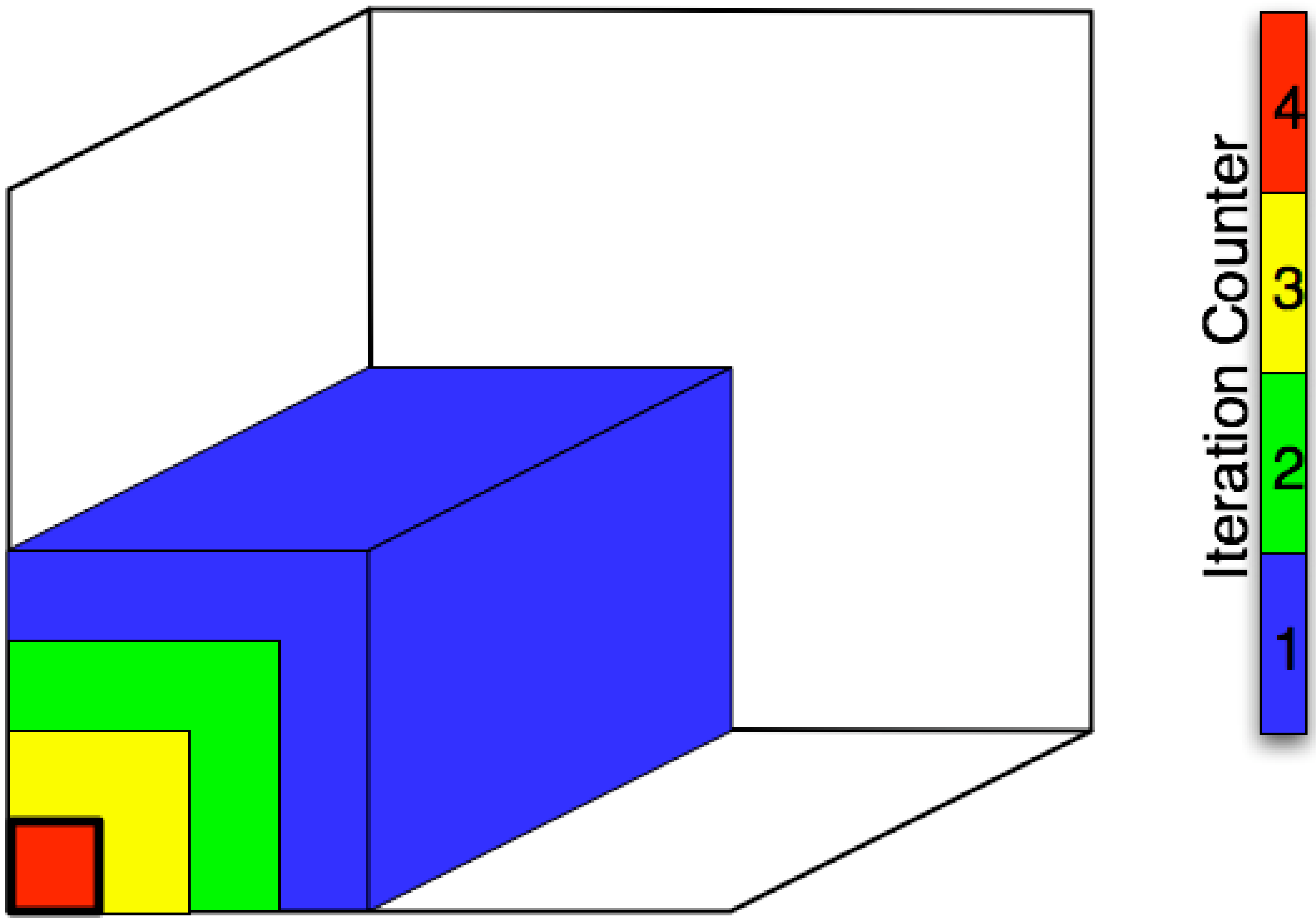
Cache Conscious - 3D Animation



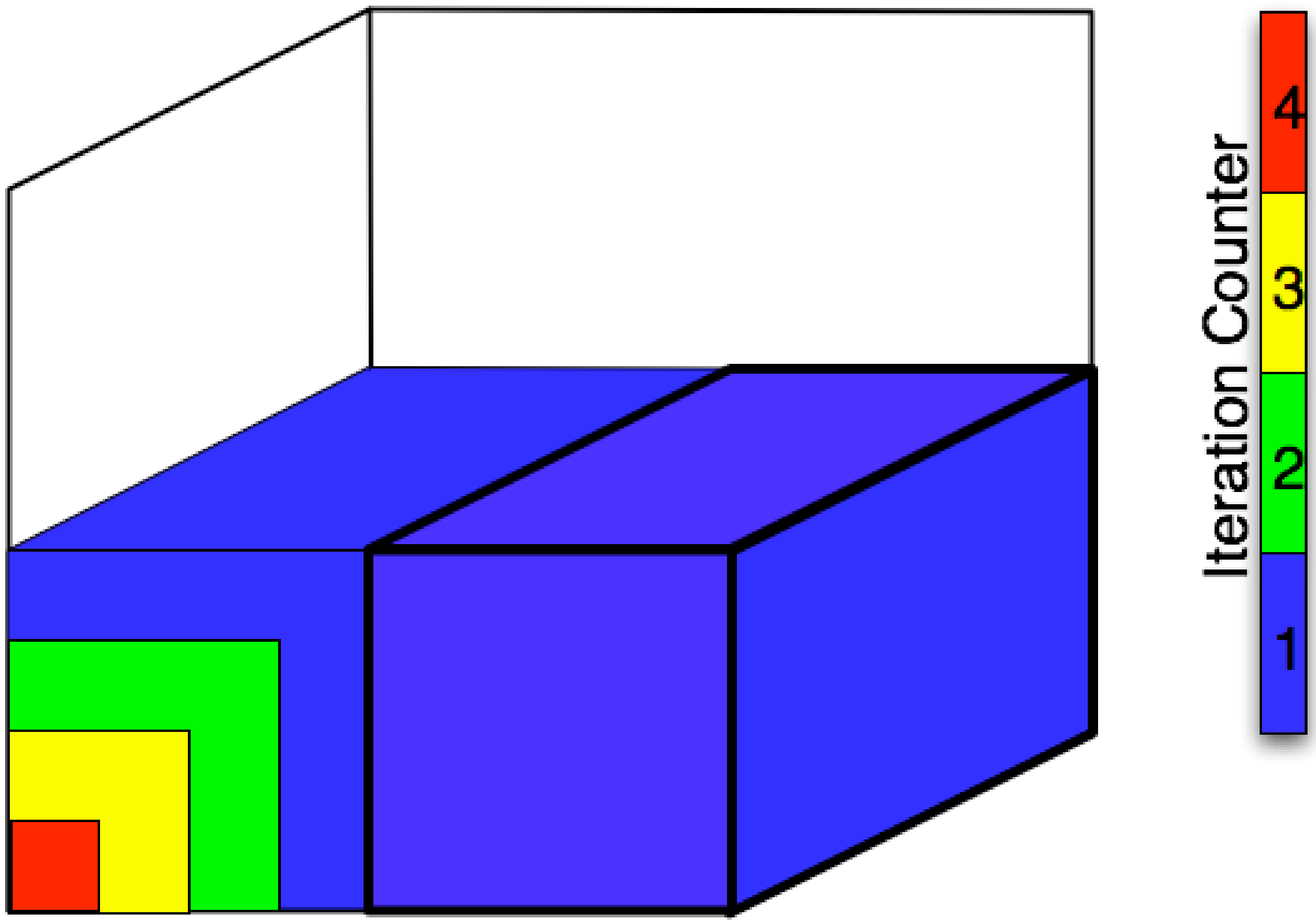
Cache Conscious - 3D Animation



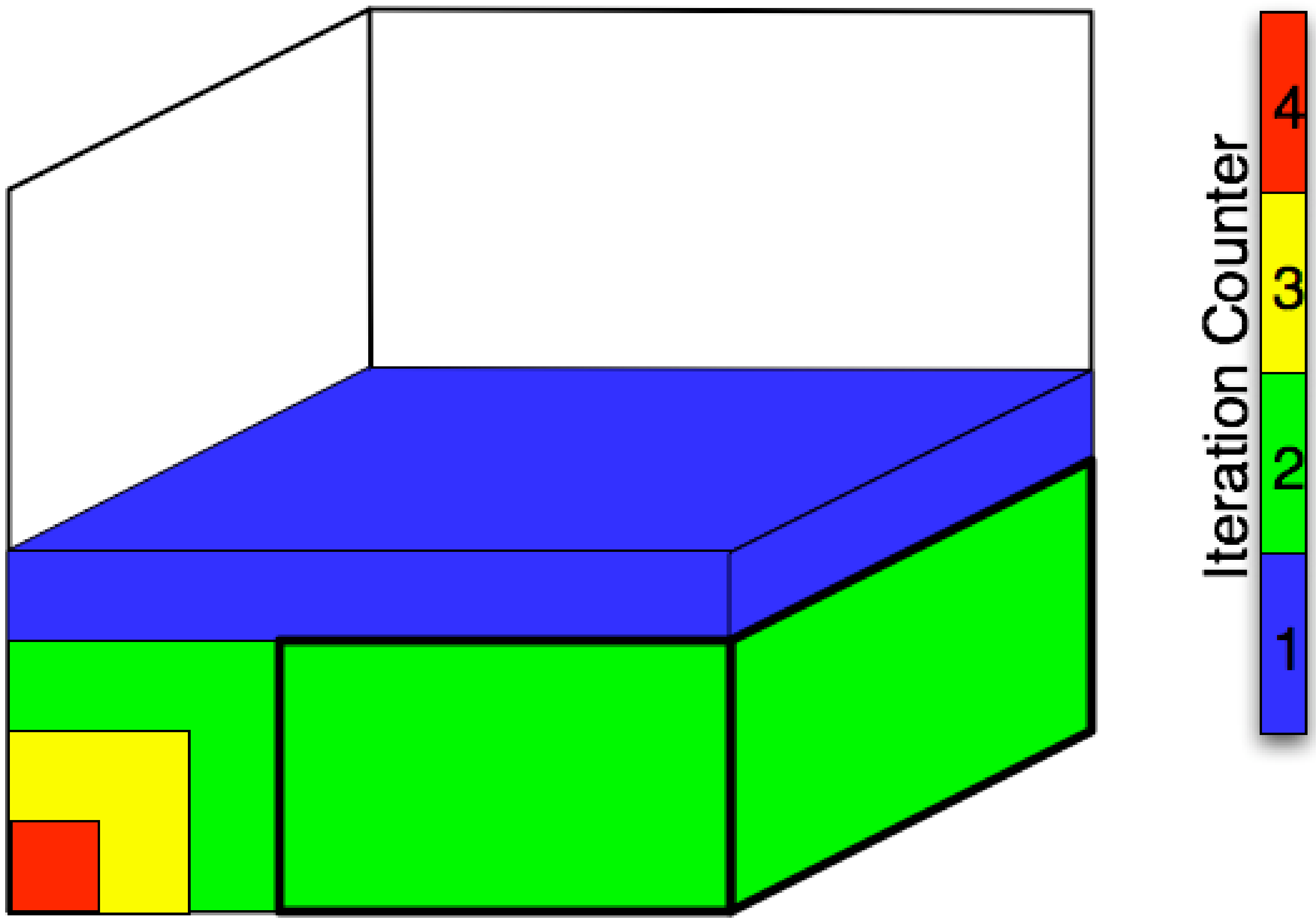
Cache Conscious - 3D Animation



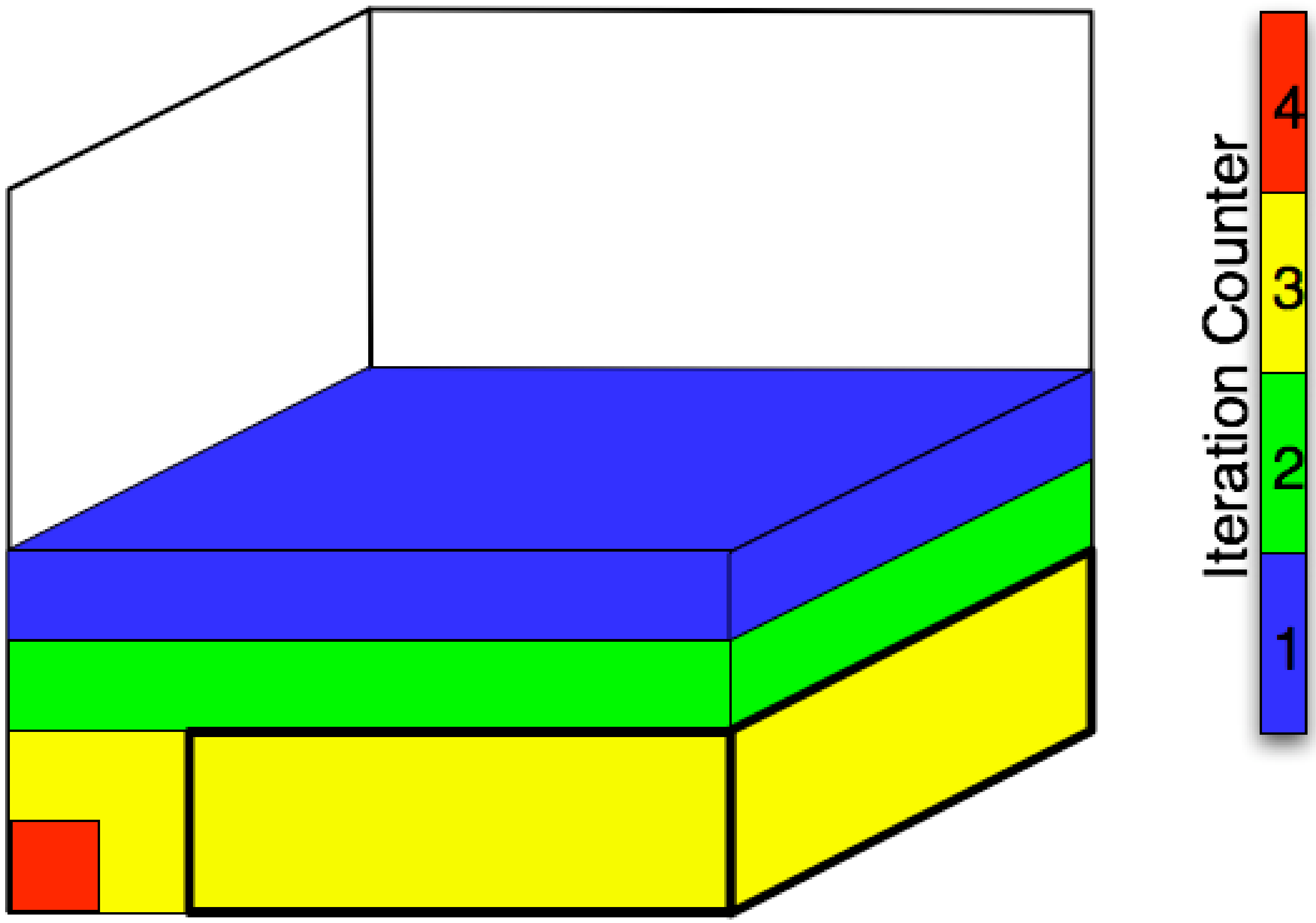
Cache Conscious - 3D Animation



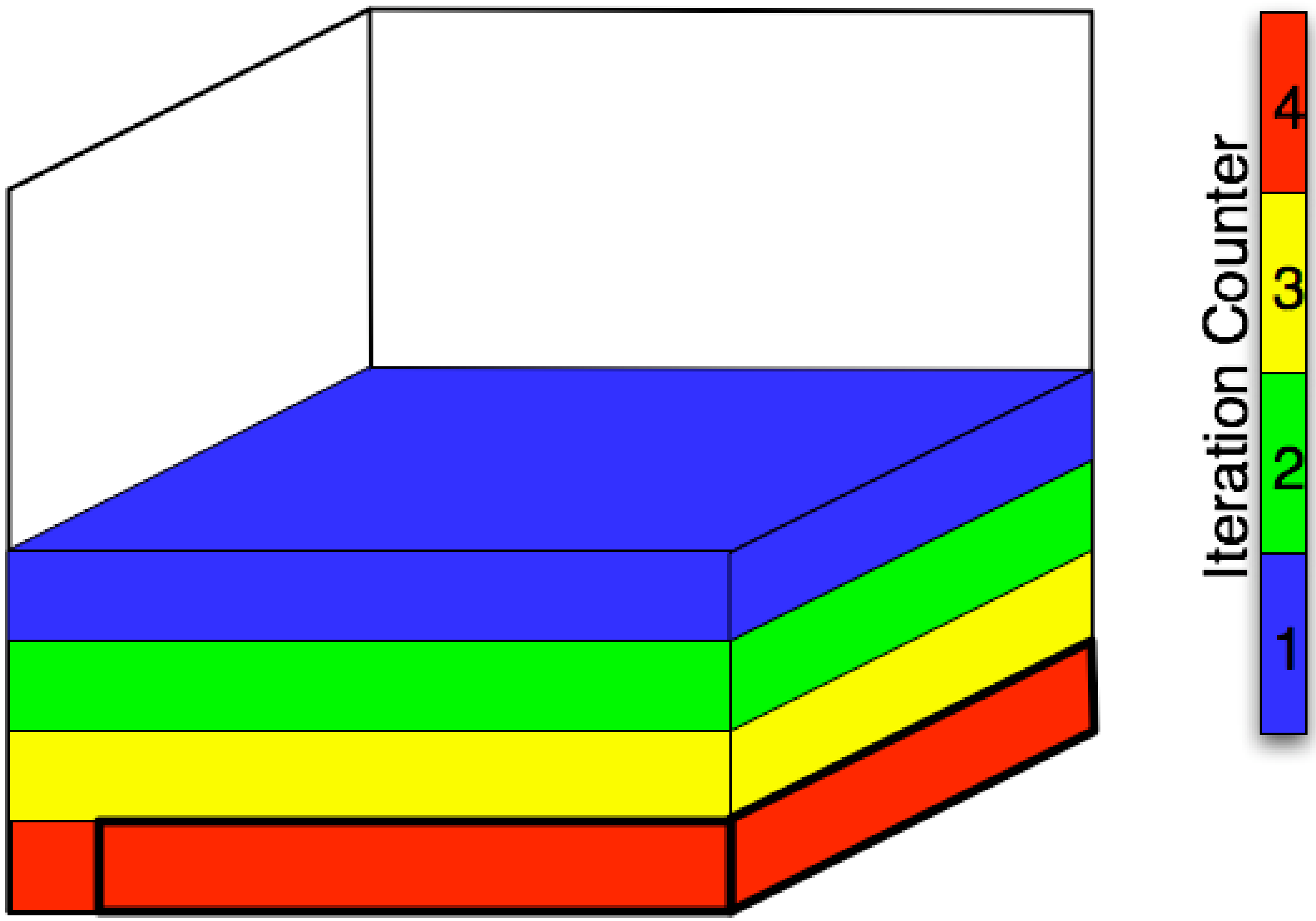
Cache Conscious - 3D Animation



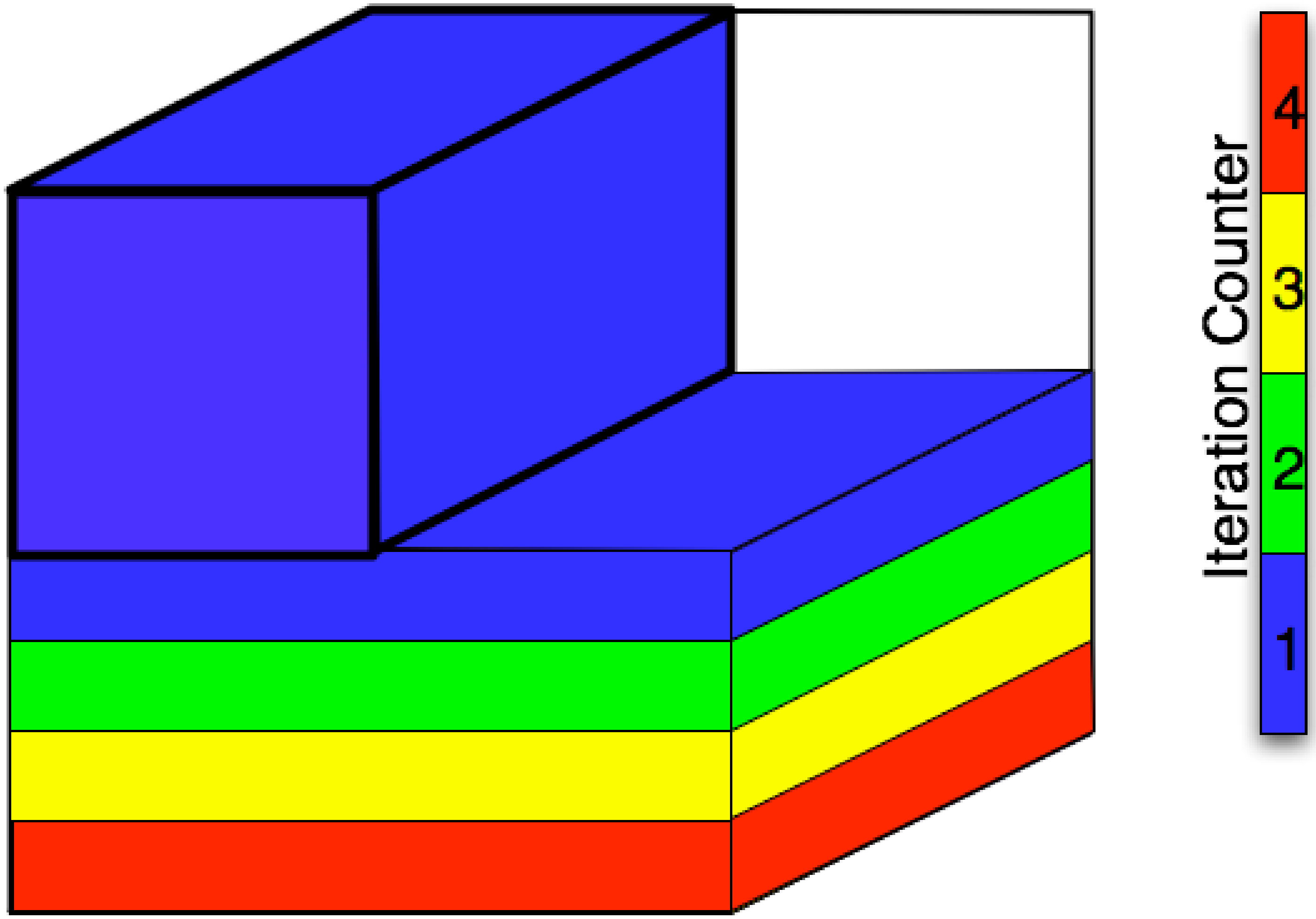
Cache Conscious - 3D Animation



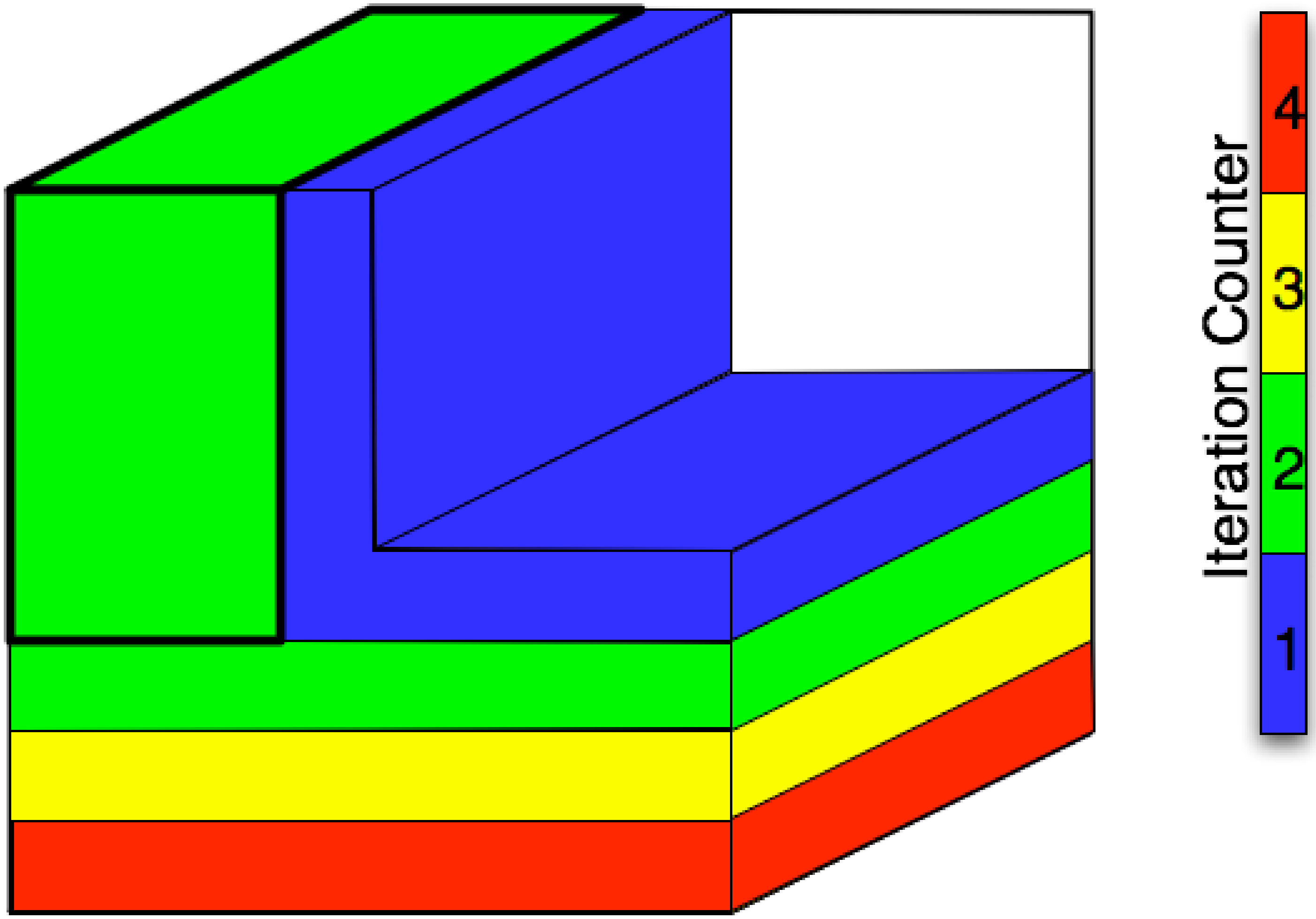
Cache Conscious - 3D Animation



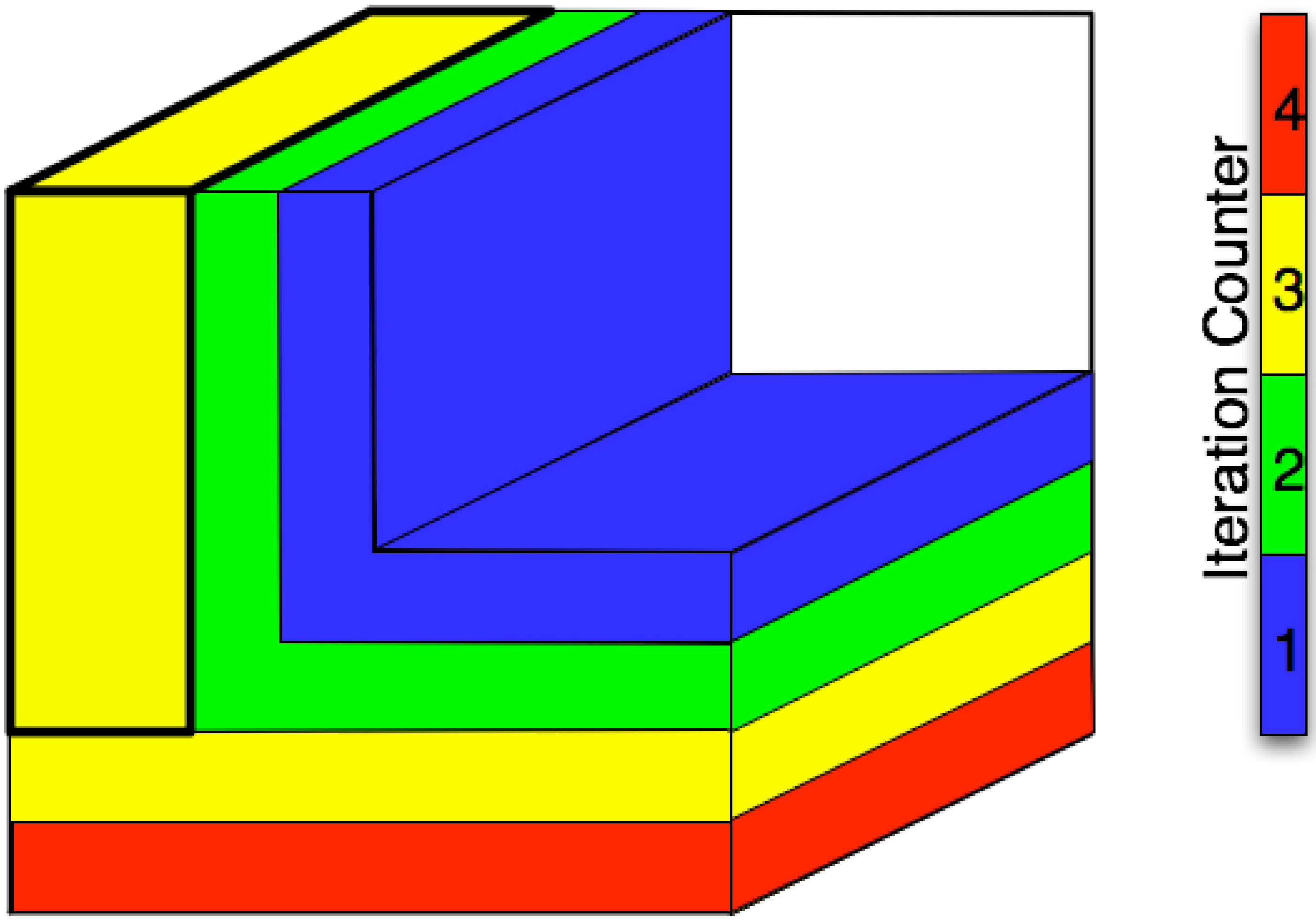
Cache Conscious - 3D Animation



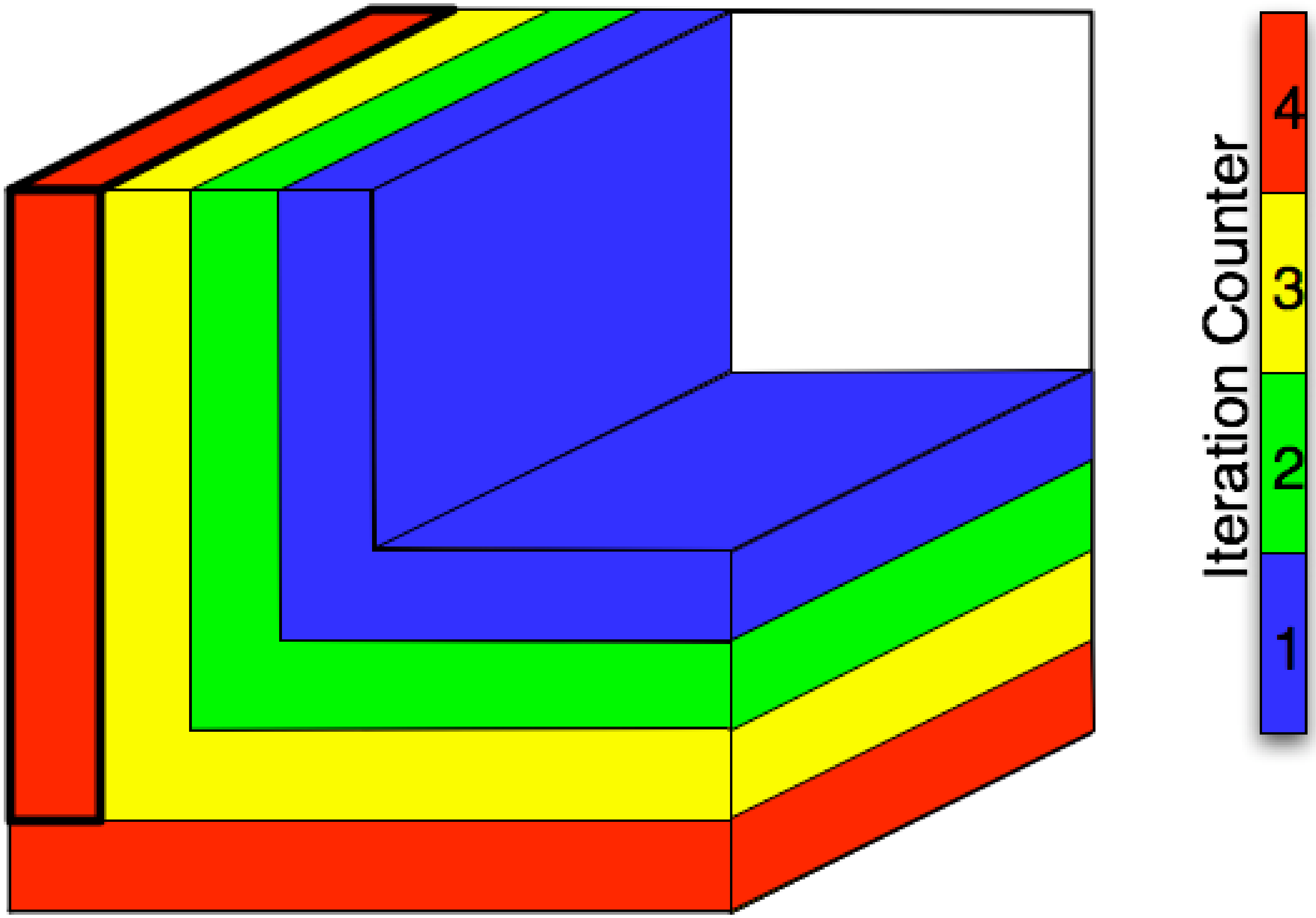
Cache Conscious - 3D Animation



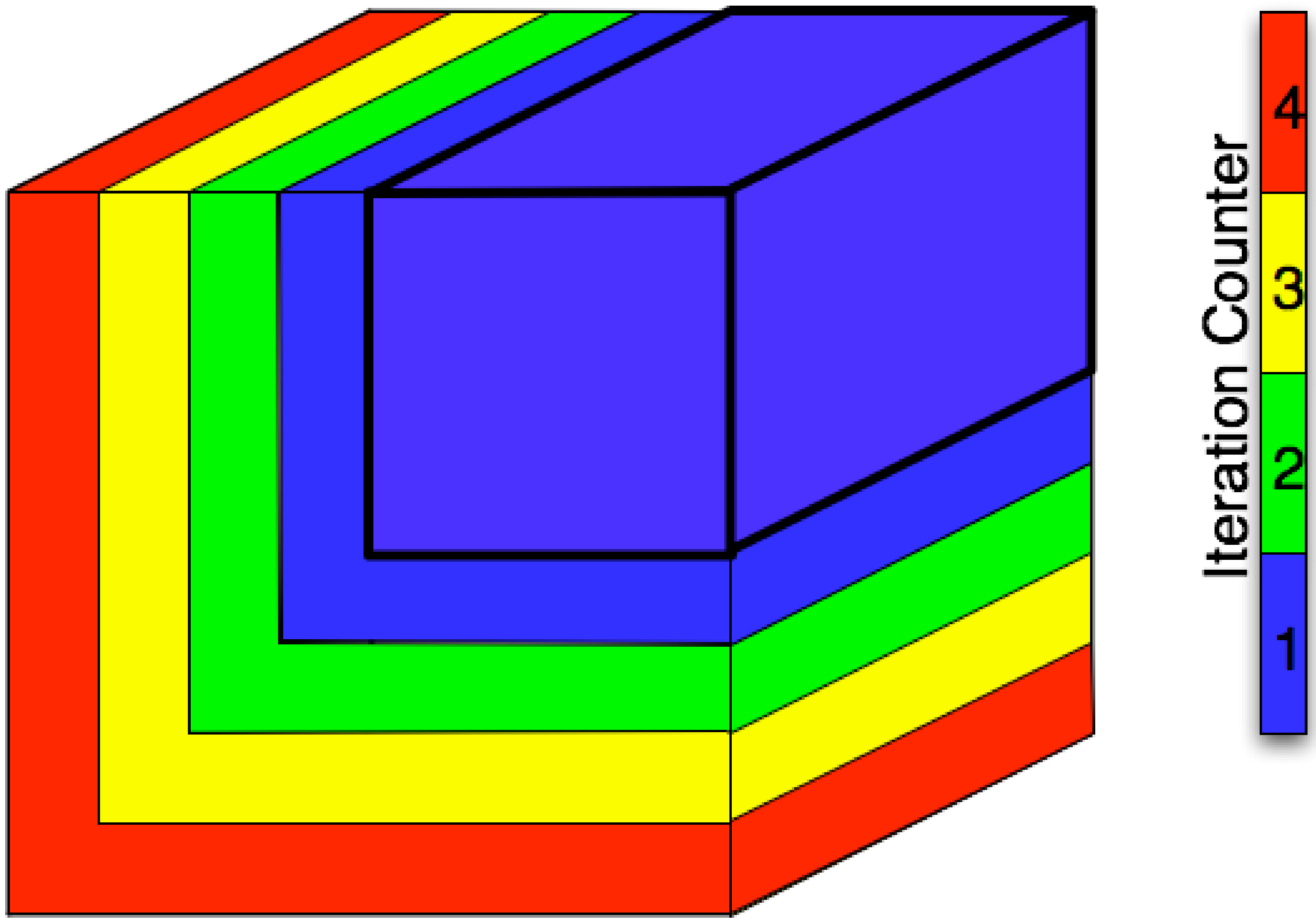
Cache Conscious - 3D Animation



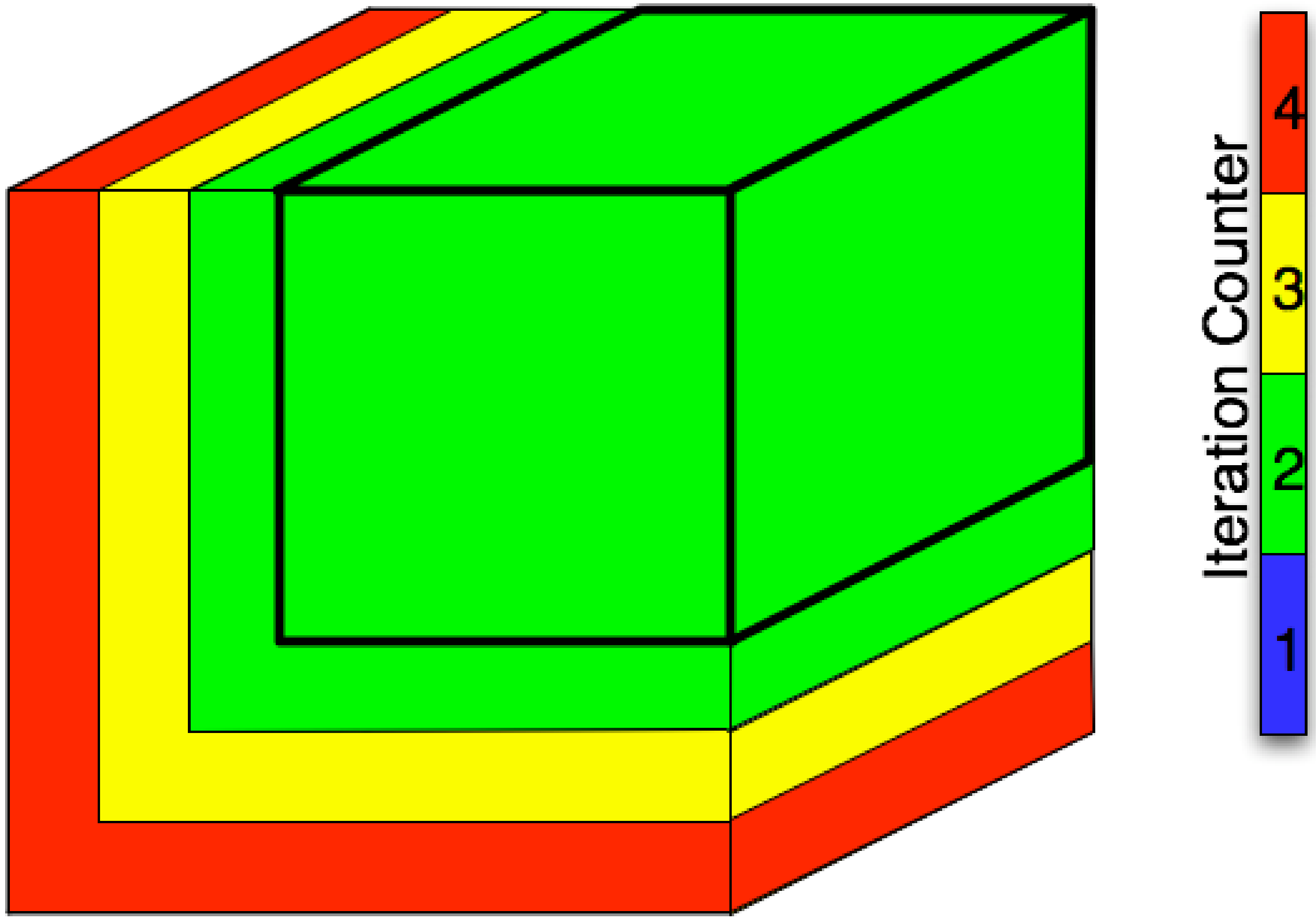
Cache Conscious - 3D Animation



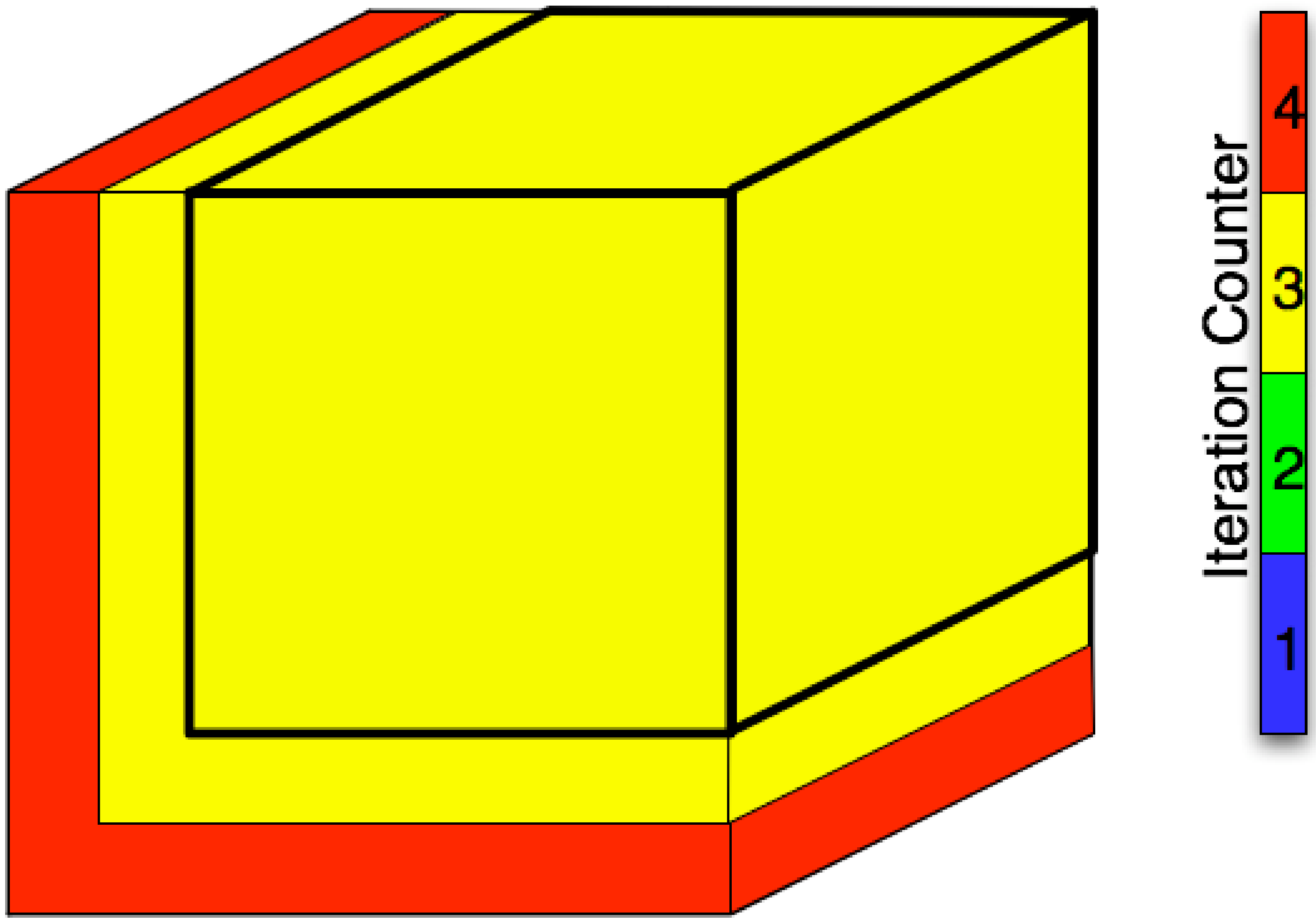
Cache Conscious - 3D Animation



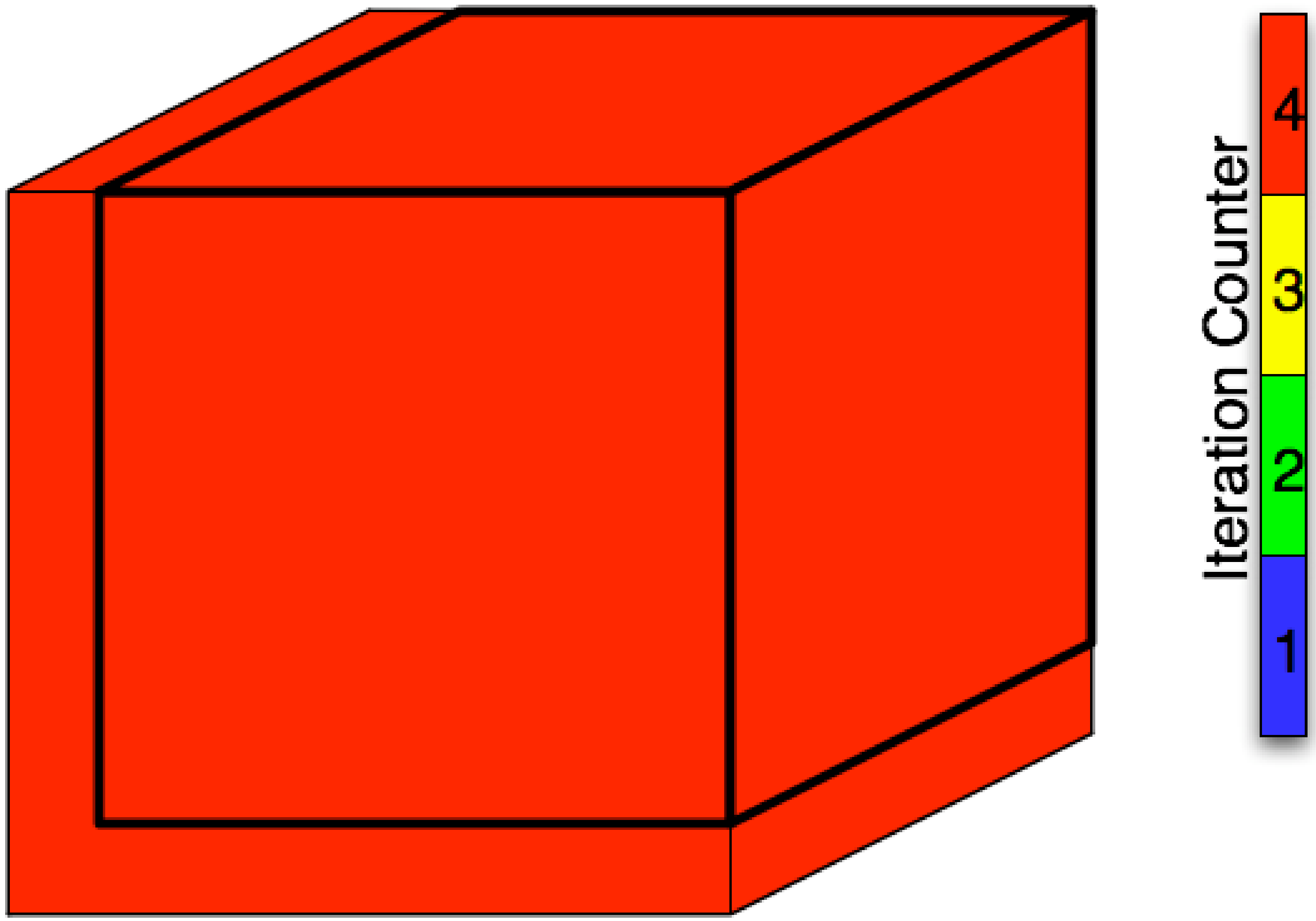
Cache Conscious - 3D Animation



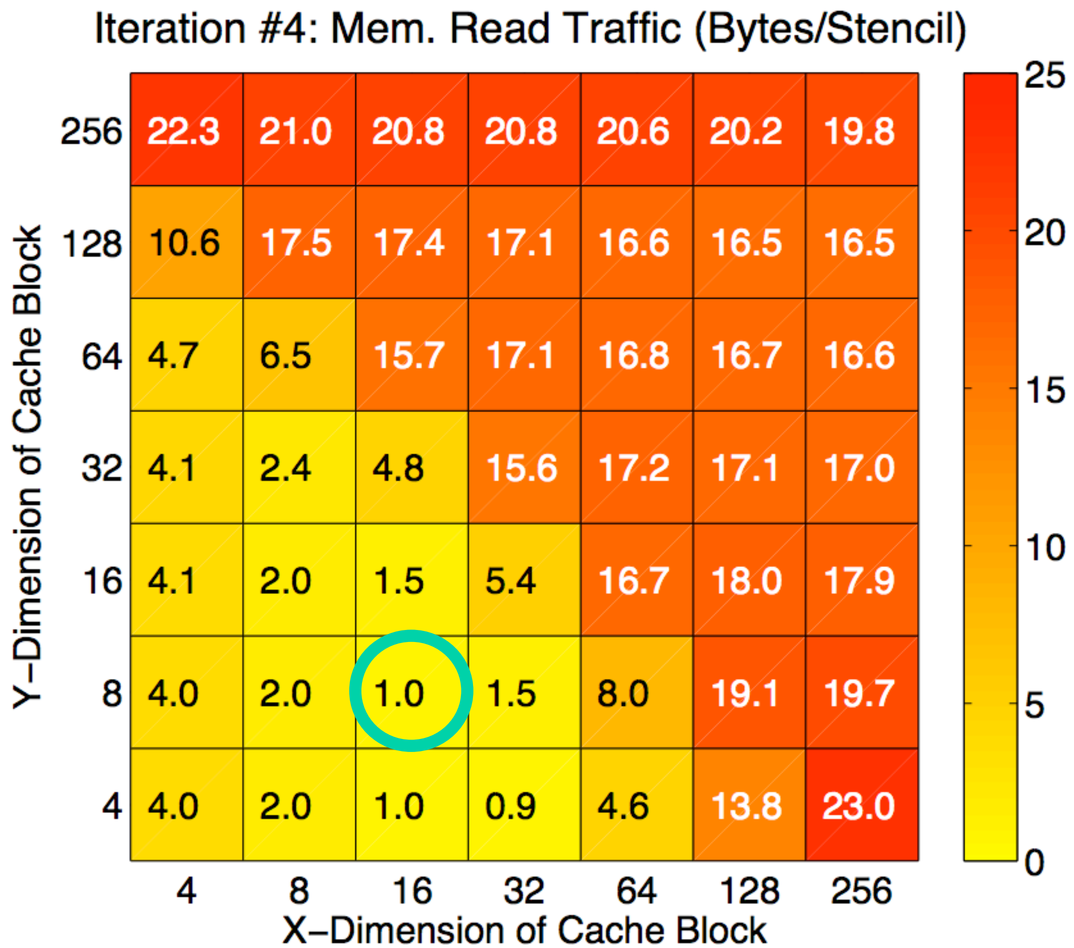
Cache Conscious - 3D Animation



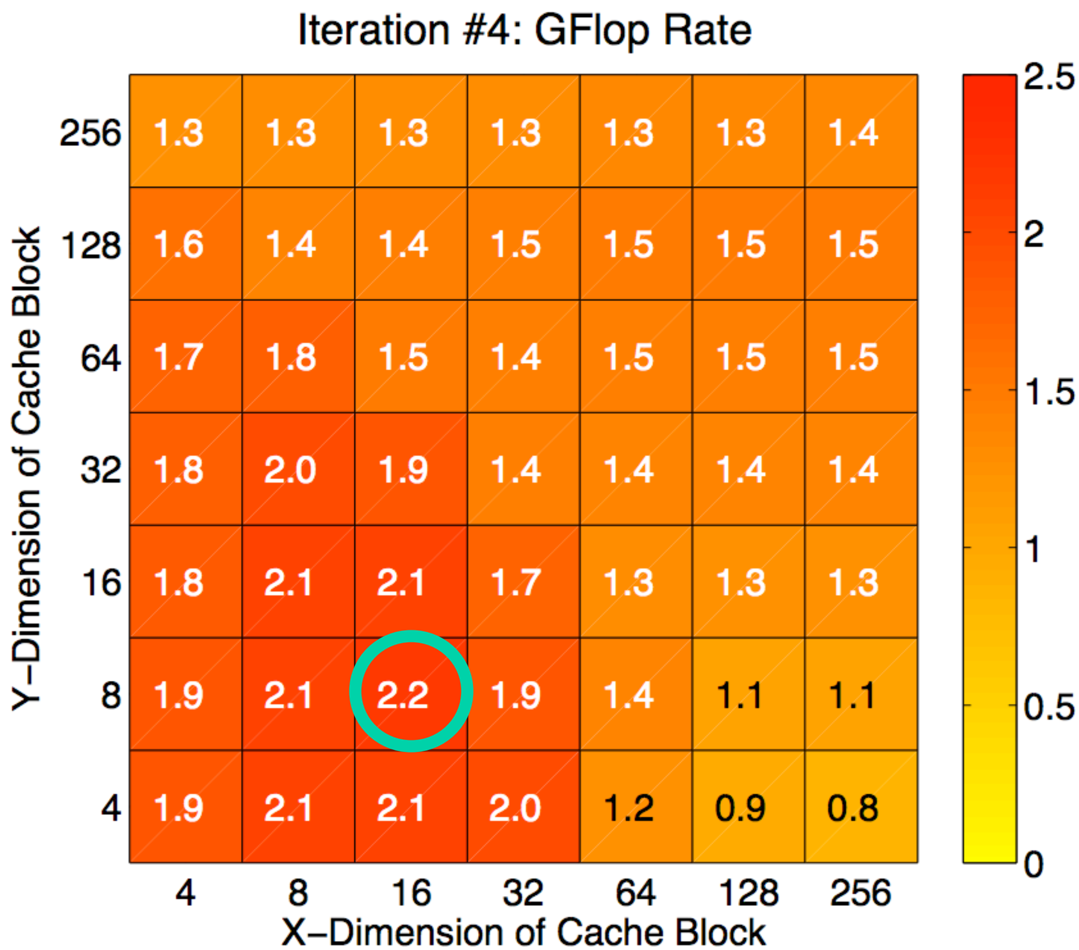
Cache Conscious - 3D Animation



Cache Conscious - Optimal Block Size Search

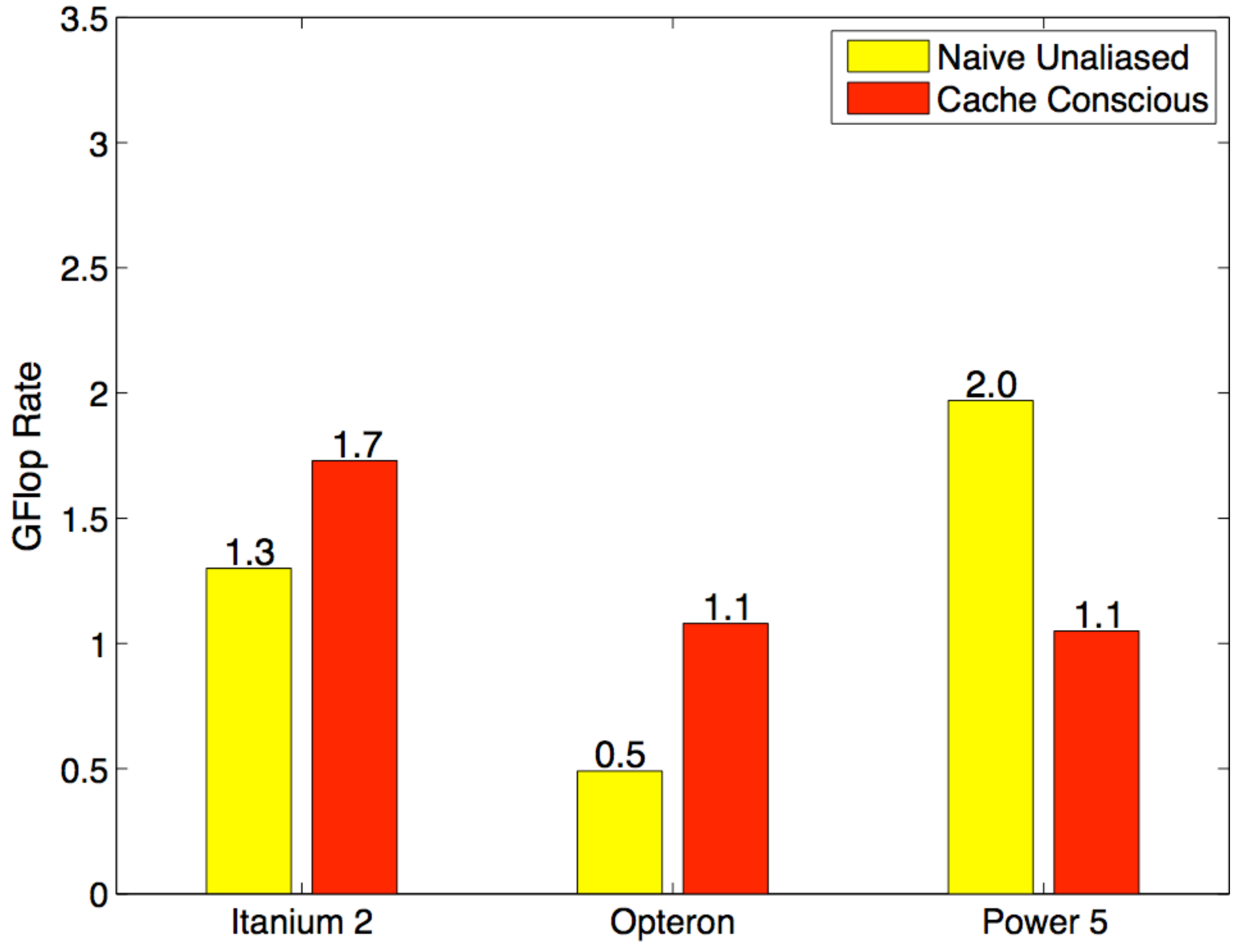


Cache Conscious - Optimal Block Size Search



- Reduced memory traffic does correlate to higher GFlop rates

Cache Conscious Performance



- Cache conscious measured with optimal block size on each platform
- Itanium 2 and Opteron both improve

Opt. Strategy #3: Cache Conscious on Cell

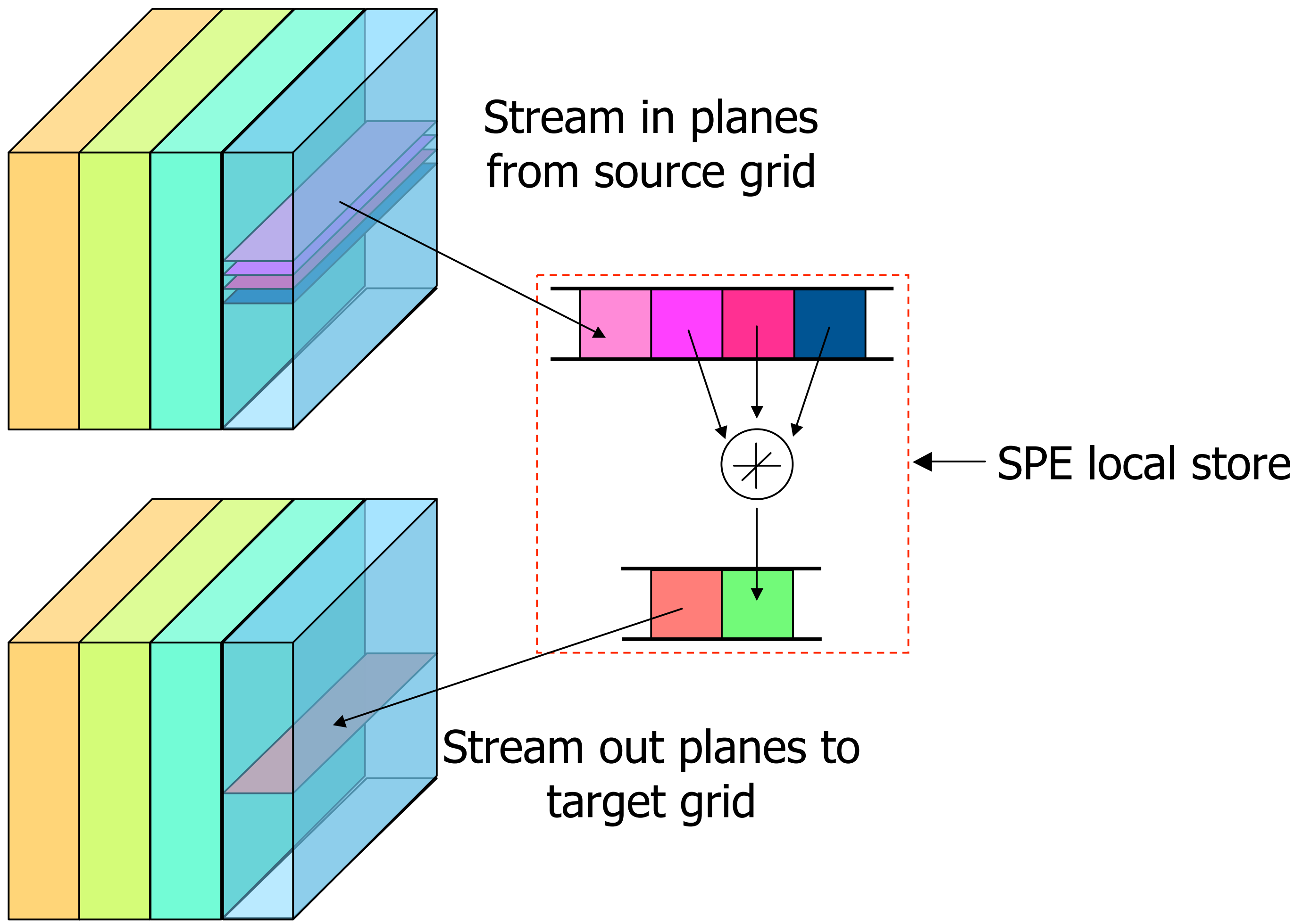
- Two software techniques
  - *Cache oblivious* algorithm recursively subdivides
  - *Cache conscious* has an explicit block size
    - Easier to visualize
    - Tunable block size
    - No recursion stack overhead
- Two hardware techniques
  - Cache managed by hw
  - Local store managed by sw
    - Eliminate extraneous reads/writes

	Hardware: Cache (Implicit)	Hardware: Local Store (Explicit)
Software: Conscious (Explicit)	Cache Conscious	Cache Conscious on Cell
Software: Oblivious (Implicit)	Cache Oblivious	N/A

Cell Processor

- PowerPC core that controls 8 simple SIMD cores ("SPE"s)
- Memory hierarchy consists of:
  - Registers
  - Local memory
  - External DRAM
- Application *explicitly* controls memory:
  - Explicit DMA operations required to move data from DRAM to each SPE's local memory
  - Effective for predictable data access patterns
- Cell code contains more low-level intrinsics than prior code

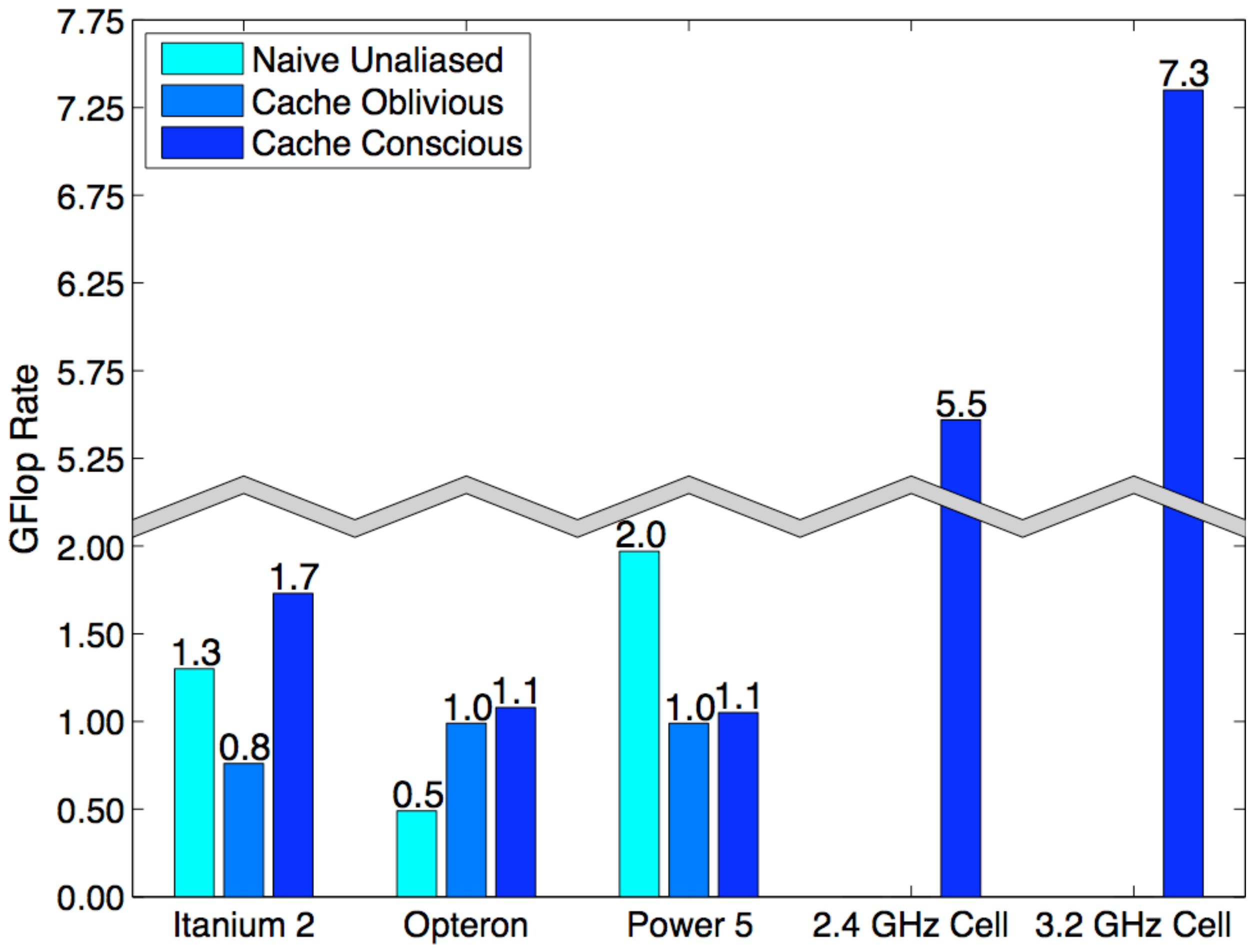
Cell Local Store Blocking



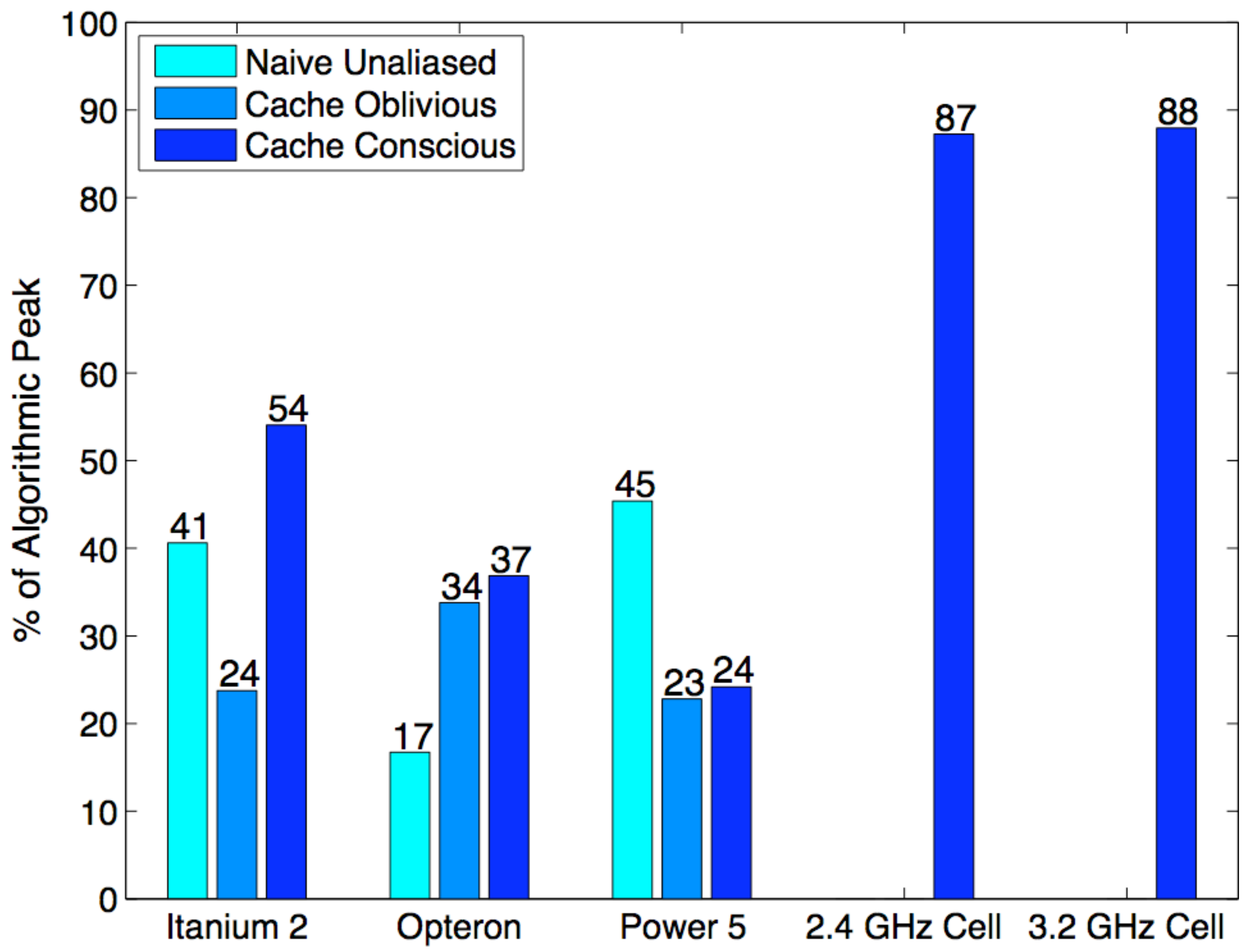
Excellent Cell Processor Performance

- Double-Precision (DP) Performance: 7.3 GFlops/s
- DP performance still relatively weak
  - Only 1 floating point instruction every 7 cycles
  - Problem becomes computation-bound when cache-blocked
- Single-Precision (SP) Performance: 65.8 GFlops/s!
  - Problem now memory-bound even when cache-blocked
- If Cell had better DP performance or ran in SP, could take further advantage of cache blocking

Summary - Computation Rate Comparison



Summary - Algorithmic Peak Comparison



Stencil Code Conclusions

- Cache-blocking performs better when explicit
  - But need to choose right cache block size for architecture
- Software-controlled memory boosts stencil performance
  - Caters memory accesses to given algorithm
  - Works especially well due to predictable data access patterns
- Low-level code gets closer to algorithmic peak
  - Eradicates compiler code generation issues
  - Application knowledge allows for better use of functional units