PARLab Parallel Boot Camp

Shared Memory Programming with Threads

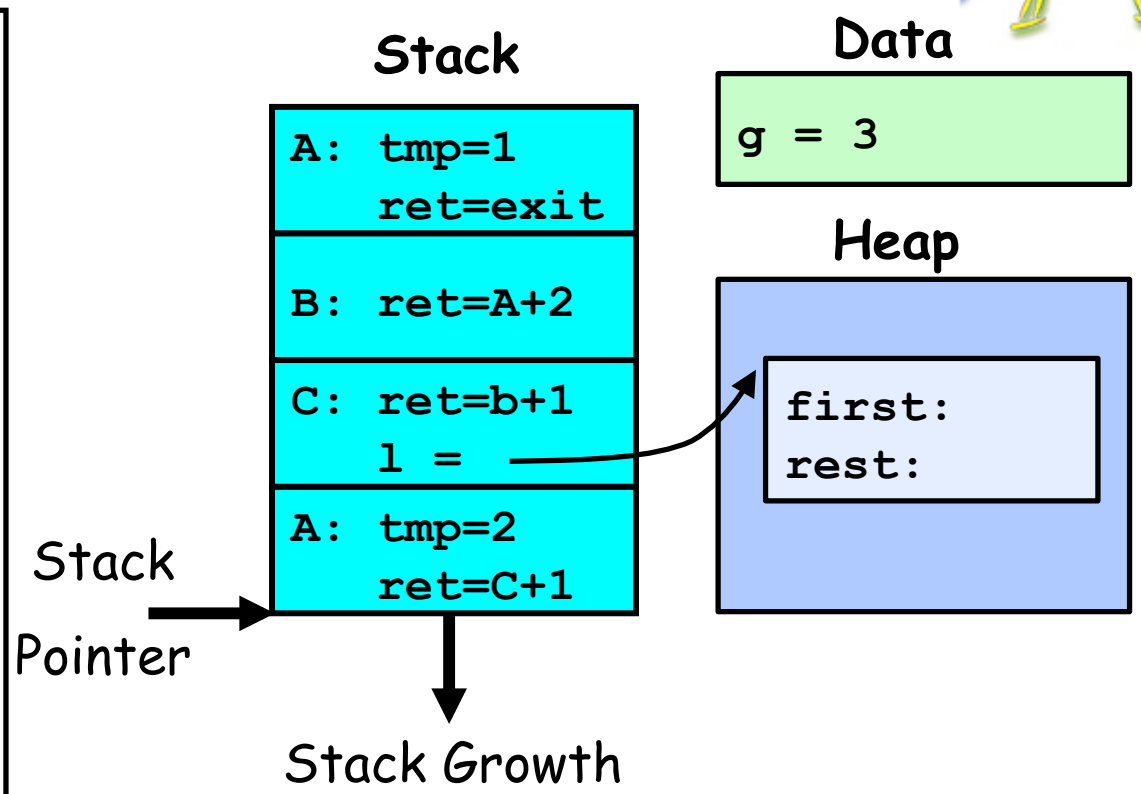
Kathy Yelick
Electrical Engineering and Computer Sciences
University of California, Berkeley
NERSC, Lawrence Berkeley National Laboratory

Outline of Shared Memory Lesson

- General concepts around threads
  - Understand what happens at runtime with threads
  - Common sources of errors
  - Common sources of performance problems
- Specific shared memory programming models
  - POSIX Threads (Pthreads): rest of this lecture
  - OpenMP: Tim Mattson
  - Thread Building Blocks (TBB): Michael Wrinn
- Continues bottom-up progression from morning:
  - Pthreads is lower level for performance experts

Sequential Runtime Structures

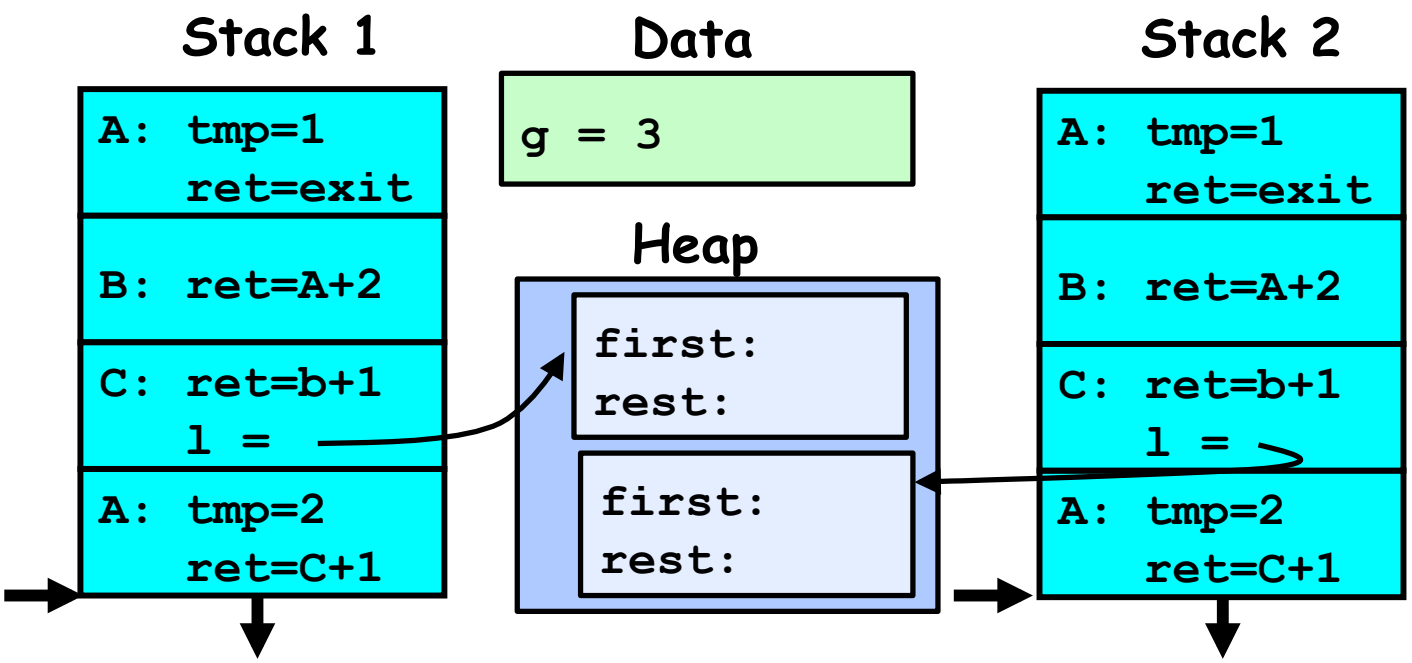
```
static int g = 3;
struct node {
  char first;
  struct node*
     rest;
};
A(int tmp) {
  if (tmp<2) B();
  printf(tmp);
}
B() {
  C();
}
C() {
  struct node *l =
    malloc (…)
  A(2);
}
A(1);
```



- Data holds global variables
- Stack holds temporary results
- Heap holds dynamically allocated objects (malloc'd)

What is a Thread?

- A *thread* is a stream of instructions that executes sequentially
- Threads may execute concurrently with one another
- E.g., fork another instance of **`A(1)`**
- Operating system schedules threads to run, so even on one processor, the may appear concurrent



How Threads are Exposed

- cobegin/coend

```
cobegin
    job1(a1);
    job2(a2);
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- future

```
v = future(job1(a1));
… = …v…;
```

- Future expression evaluated in parallel
- Attempt to use return value will wait

- Cobegin cleaner than fork, but fork is more general
- Threads expressed in the code may not turn into independent computations
  - Only create threads if processors idle
  - Example: Thread-stealing runtimes such as cilk

Overview of POSIX Threads

- POSIX: *Portable **O**perating **S**ystem **I**nterface for UNI**X***
  - Interface to Operating System utilities
- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- Pthreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking POSIX Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Simple Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

E.g., compile using gcc -lpthread

Shared Data and Threads

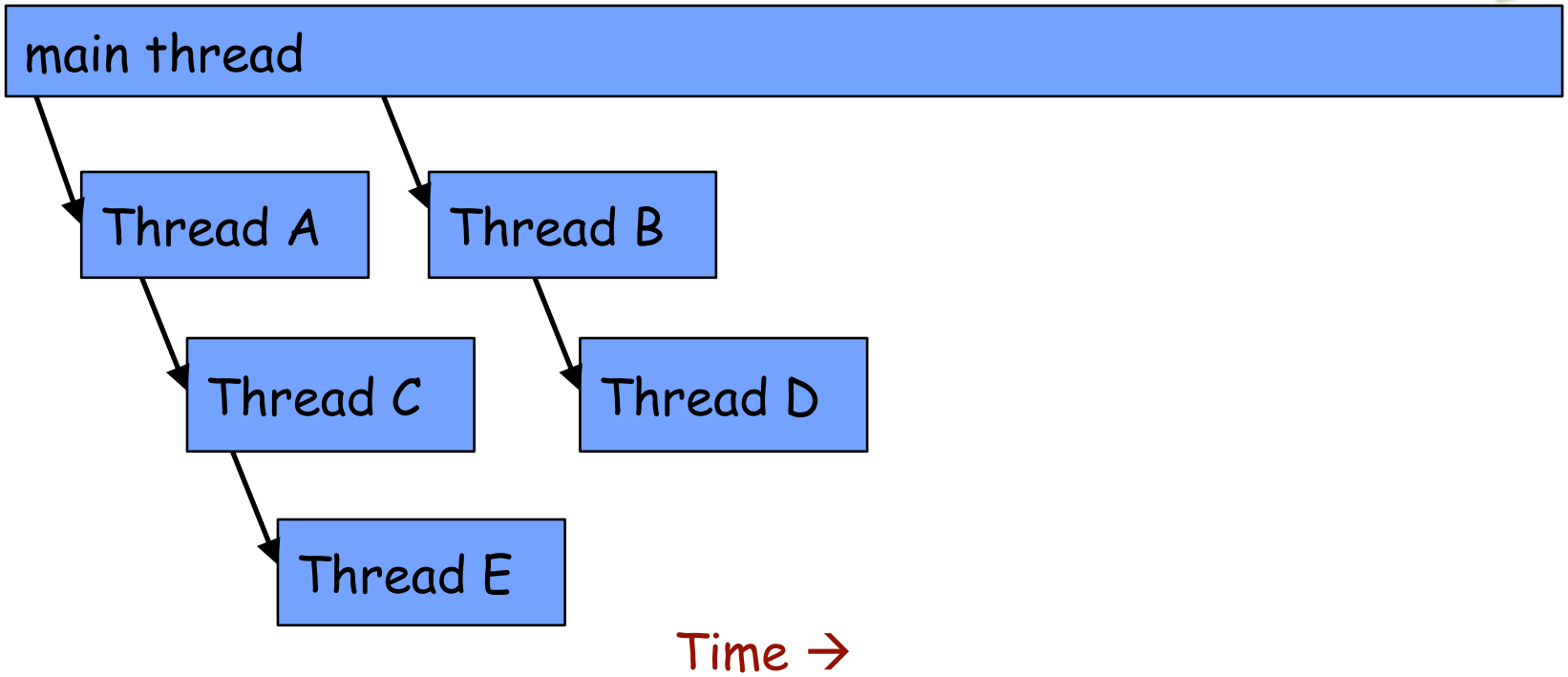
- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large "thread data" struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";

pthread_create(&thread1, NULL,
     print_fun,(void*) message);
```

Thread Scheduling



- Once created, when will a given thread run?
- It is up to the Operating System, but it will run eventually, even if you have more threads than cores
  - Not true with all thread libraries, e.g., user level thread packages
- Can provide hints/control in some case, or affinity
  - E.g., create exactly P threads and assign to P cores

Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (…, update_stuff, …,
                      &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_stuff should have a significant amount of work
- Common Performance Pitfall: Too many threads
  - The cost of creating a thread is 10s of thousands of cycles on modern architectures
  - Solution: Thread blocking: use a small # of thread, often equal to the number of cores/processors or hardware threads

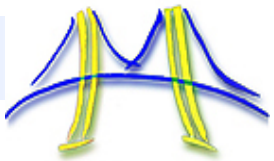
Some More Pthread Functions

- pthread_yield();
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.

- pthread_t me; me = pthread_self();
  - Allows a pthread to obtain its own identifier pthread_t thread;

- pthread_detach(thread);
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance. For more information consult the library or the man pages, e.g., man -k pthread..

Setting Attribute Values



- Once an initialized attribute object exists, changes can be made. For example:
  - To change the stack size for a thread to 8192 (before calling pthread_create), do this:
    - » pthread_attr_setstacksize(&my_attributes, (size_t)8192);
  - To get the stack size, do this:
    - » size_t my_stack_size;
      pthread_attr_getstacksize(&my_attributes, &my_stack_size);
- Other attributes:
  - Detached state – set if no other thread will use pthread_join to wait for this thread (improves efficiency)
  - Guard size – use to protect against stack overfow
  - Inherit scheduling attributes (from creating thread) – or not
  - Scheduling parameter(s) – in particular, thread priority
  - Scheduling policy – FIFO or Round Robin
  - Contention scope – with what threads does this thread compete for a CPU
  - Stack address – explicitly dictate where the stack is located
  - Lazy stack allocation – allocate on demand (lazy) or all at once, "up front"

Data Race Example

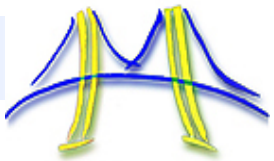
```
static int s = 0;
```

```
Thread 1
  for i = 0, n/2-1
    s = s + 1
```

```
Thread 2
  for i = n/2, n-1
    s = s + 1
```

- Common Correctness pitfall: race conditions
- A race condition or data race occurs when:
  - Two threads access the same variable, and at least one does a write
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Data Race Example



```
static int s = 0;
```

Thread 1 (on core 1)	Thread 2 (on core 2)
Read s into R1	Read s into R2
Inc R1	Inc R2
Write R1 to s	Write R2 to s

- Increments may be lost if both threads read simultaneously
- Need to make the read+increment+write an "atomic" operation
- How to you recognize a likely race?
- Solution: locks aka Mutexes

Basic Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

- Other languages have lexically scoped synchronization
  - » analogous to cobegin/coend vs. fork and join tradeoff
  - » E.g., Java synchronized methods

Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Correctness Pitfalls with Locks

- Correctness pitfall: locks cover too small a region

```
acquire (a)
tmp = s
release (a)
tmp++
acquire (a)
s = tmp
release (a)
```

No races, but updates can still be lost

- Correctness pitfall: Multiple locks may be held, but can lead to deadlock:

thread1	thread2
lock(a)	lock(b)
lock(b)	lock(a)

Performance Pitfalls with Locks

- Performance pitfall: critical region (code executing holding lock) is too large
  - Little or no true parallelism
  - Lock cost can go up with more contention
  - Solution: make critical regions as small as possible (but no smaller)
  - Solution: Use different locks for different data structures
- Performance pitfall: locks themselves may be expensive
  - The overhead of locking / unlocking can be high
  - Solution: roll your own spinlocks ☹

Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - » SPMD "Single Program Multiple Data"
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
  work1();
  barrier
} else { barrier }
```

- barriers are not provided in all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.

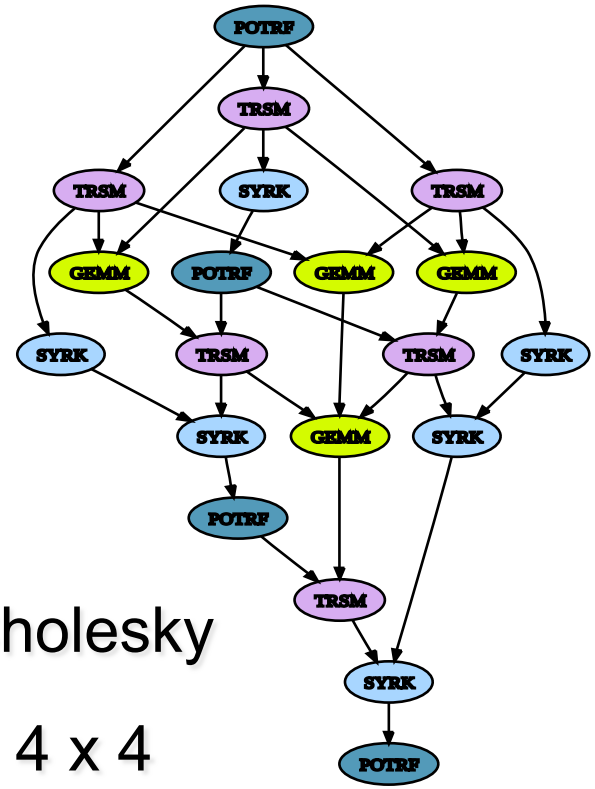
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

Performance Pitfall: too many Barriers

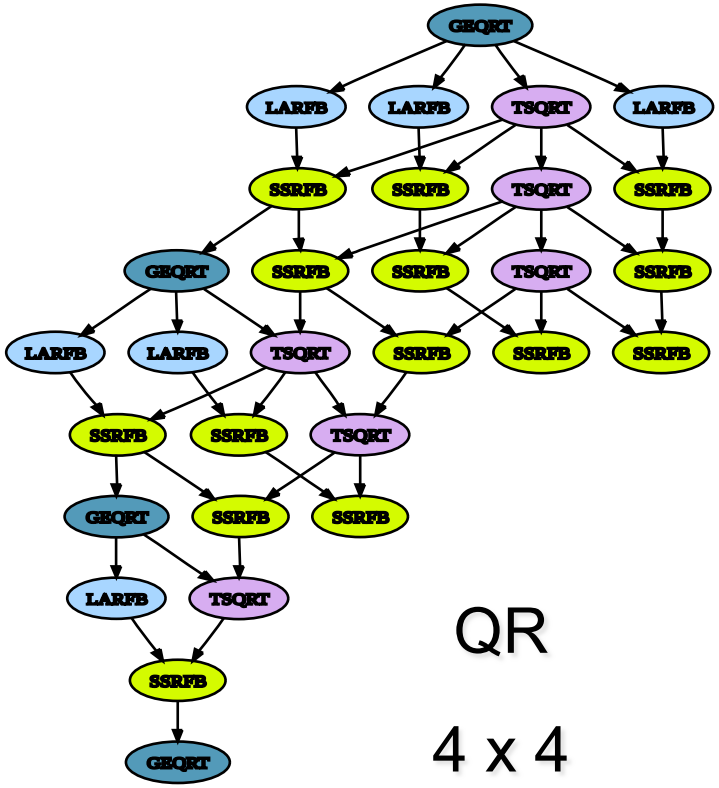
Computations as DAGs

View parallel executions as the directed acyclic graph of the computation



Cholesky

4 x 4



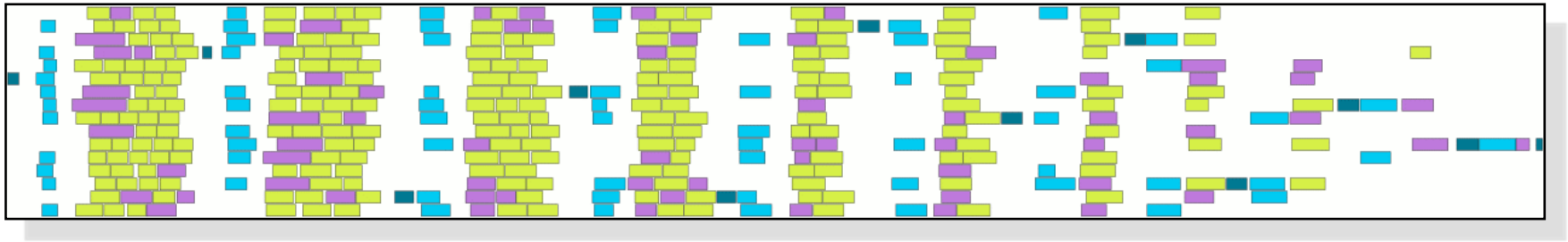
QR

4 x 4

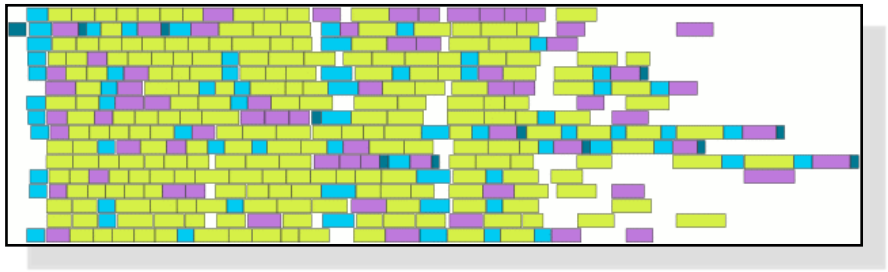
Slide source: Jack Dongarra

Illustration from PLASMA Project

Nested fork-join parallelism (e.g., Cilk, TBB)



Arbitrary DAG scheduling (e.g., PLASMA,

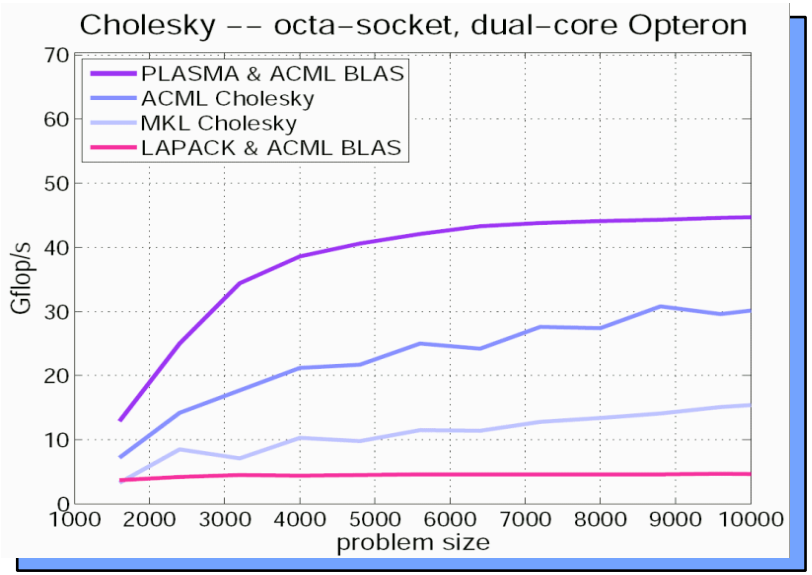


SuperMatrix)

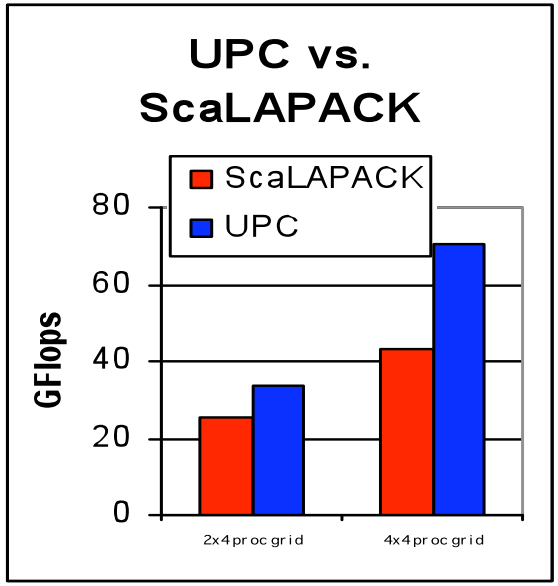
Slide source: Jack Dongarra

DAG Scheduling Outperforms Bulk-Synchronous Style

PLASMA on shared memory



UPC on partitioned memory



UPC LU factorization code adds cooperative (non-preemptive) threads for latency hiding

- New problem in partitioned memory: allocator deadlock
- Can run on of memory locally due tounlucky execution order

PLASMA by Dongarra et al; UPC LU joint with Parray Husbands

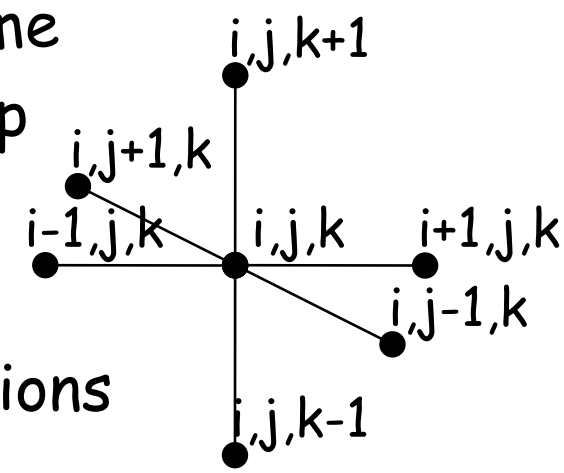
Summary of Pthreads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative

Performance Study: Stencils

- Nearest neighbor computations on structured grids (1D…ND array)
- stencils from PDEs are often a weighted linear combination

  of neighboring values
- cases where weights vary in space/time
- stencil can also result in a table lookup
- stencils can be nonlinear operators



- caveat: We only examine implementations

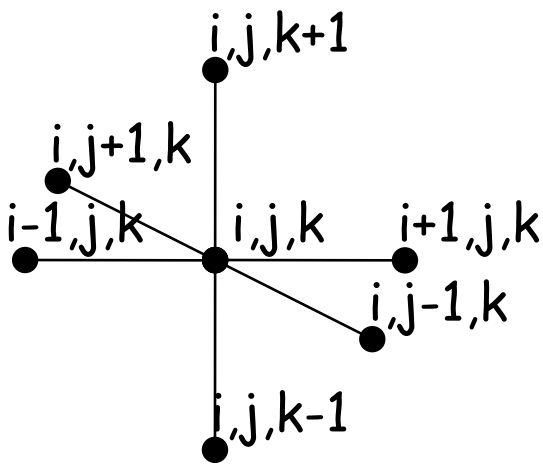
  like Jacobi's Method

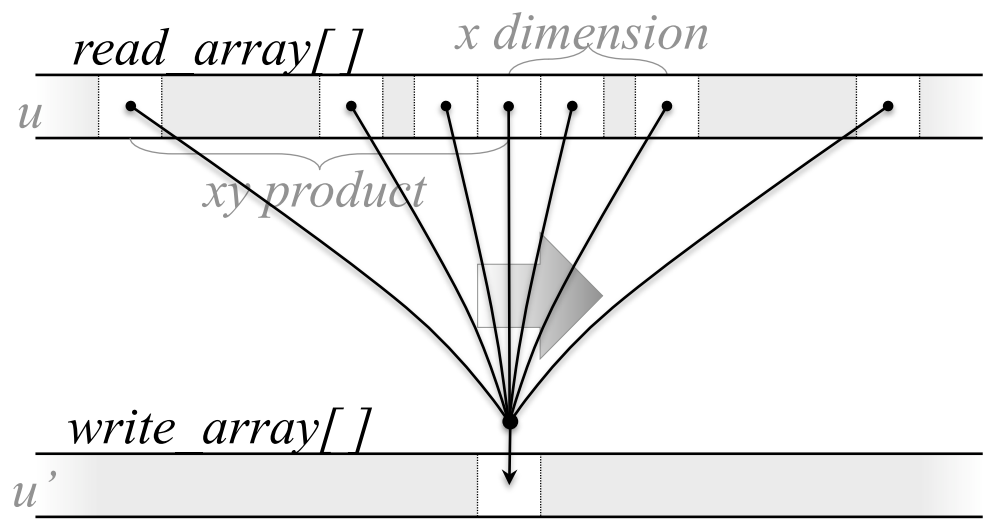
  (i.e. separate read and write arrays)

Laplacian Differential Operator

- 7-point stencil on scalar grid, produces a scalar grid

- Substantial reuse (+high working set size)
- **Memory-intensive** kernel
- Elimination of capacity misses may improve performance by **66%**



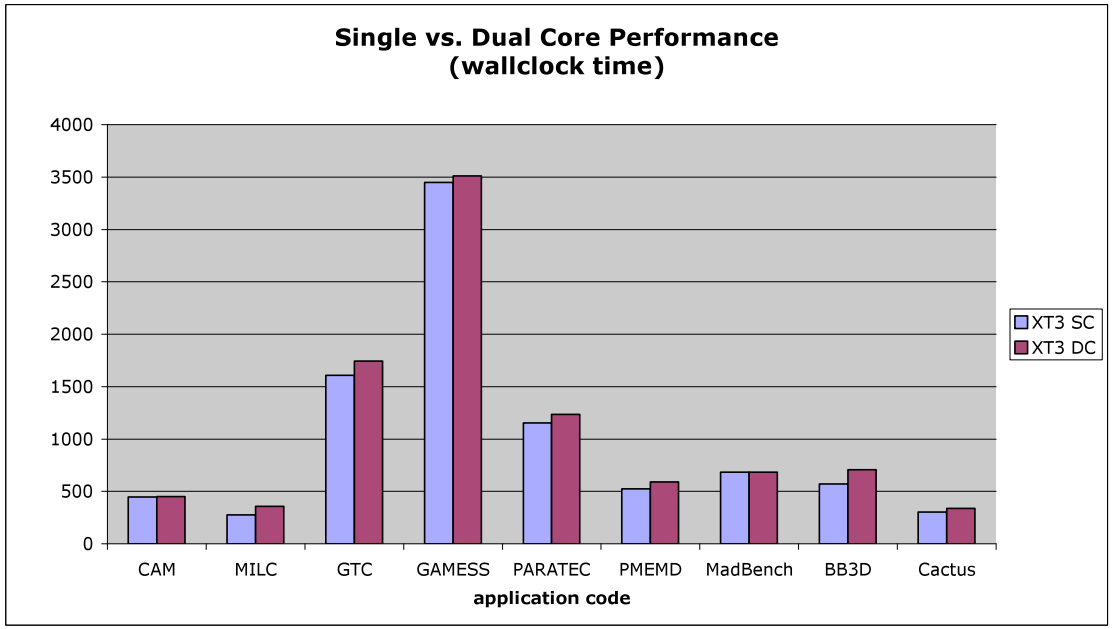


Benchmark Machines

Pitfall: Ignore Little's Law

Little's Law: required concurrency = bandwidth * latency

#outstanding_memory_fetches = bandwidth* latency



NERSC application benchmarks

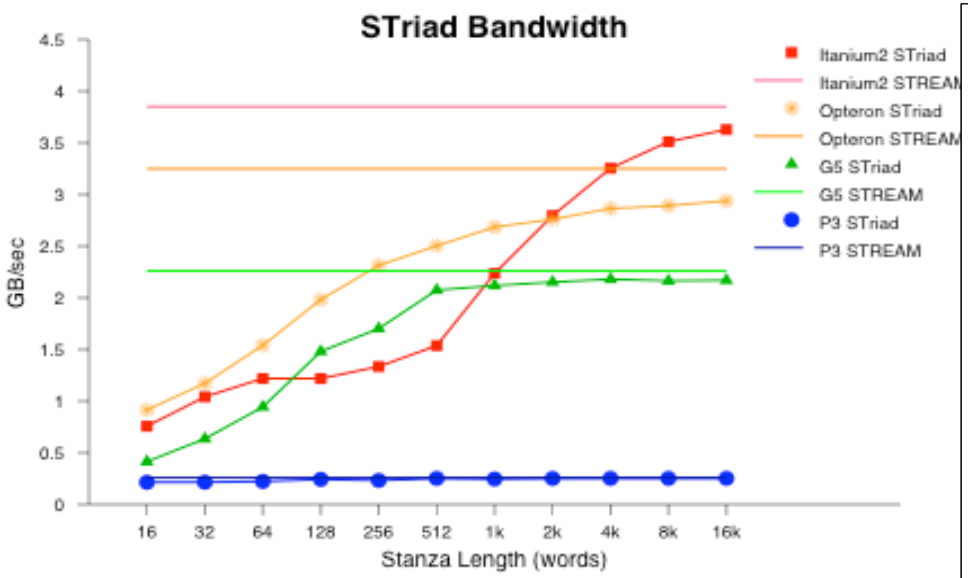
Shalf et al

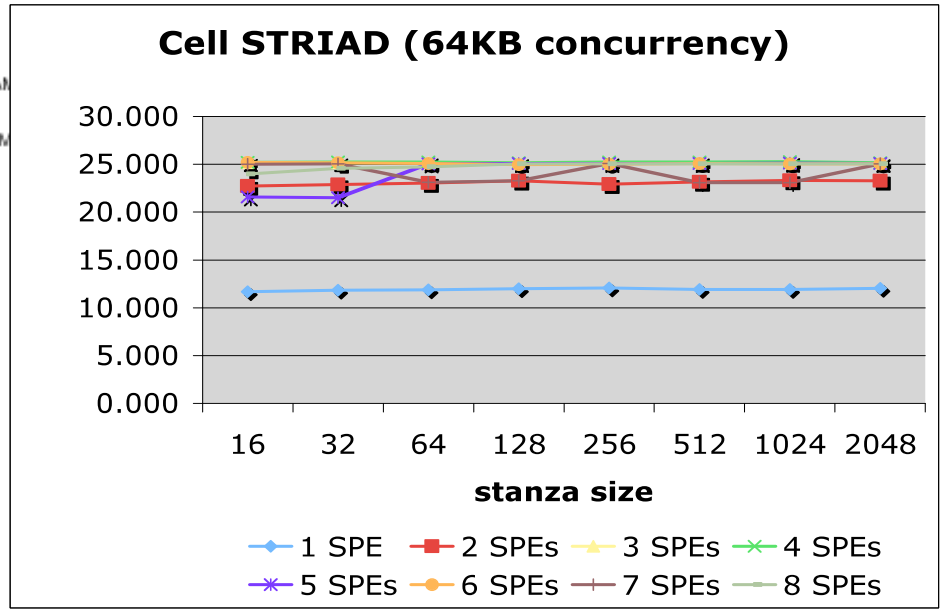
Experiment: Running on a fixed number of cores

1 core per socket vs 2 cores per socket

Only 10% performance drop from sharing (halving) bandwidth

Pitfall: Memory Latency



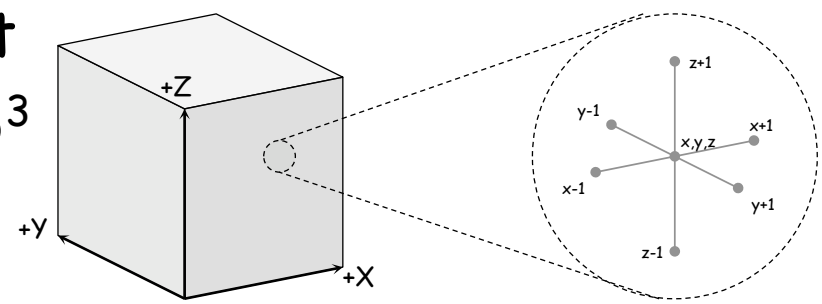


- Unit stride access is as important as cache utilization on processors that rely on hardware prefetch
  - Tiling in unit stride direction is counter-productive: improves reuse, but kills prefetch effectiveness
- Software controlled memory gives programmers more control
  - Spend bandwidth on what you use; bulk moves (DMA) hide latency

Joint work with Shoaib Kamil, Lenny Oliker,
John Shalf, Kaushik Datta

The Heat Equation Stencil

- Explicit Heat equation (Laplacian ~ $\nabla^2 F(x,y,z)$) on a regular grid
- Storage: one double per point in space
- 7-point nearest neighbor stencil
- Must:
  - read every point from DRAM
  - perform 8 flops (linear combination)
  - write every point back to DRAM
- **Just over 0.5 flops/byte (ideal)**
- Cache locality is important
- Run one problem size: 256^3

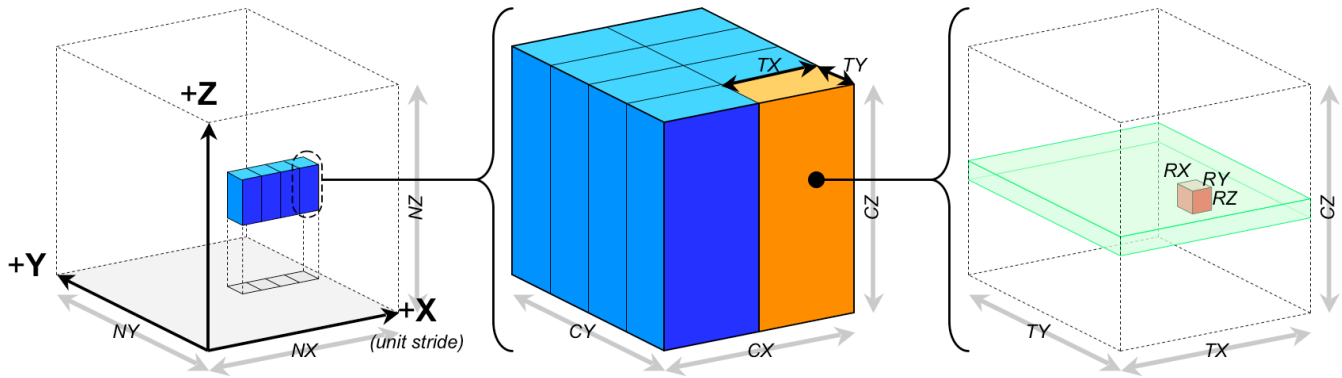


PDE grid

stencil for heat equation PDE

Tuning / Optimizations

- Strategy Engine explores a number of auto-tuning optimizations:
  - loop unrolling/register blocking
  - cache blocking
  - constant propagation / common subexpression elimination



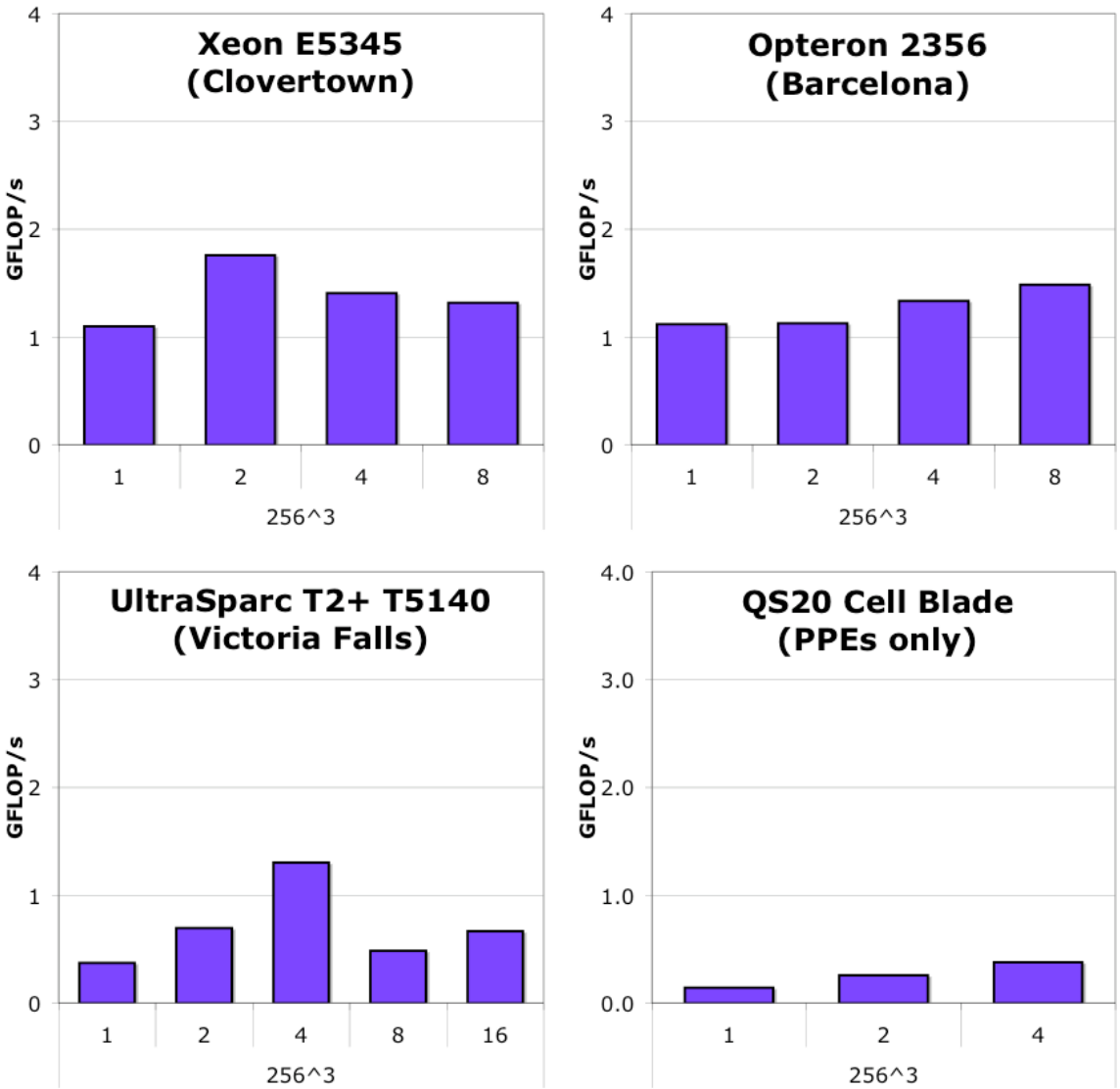
(a) Decomposition of a Node Block into a Chunk of Core Blocks

(b) Decomposition into Thread Blocks

(c) Decomposition into Register Blocks

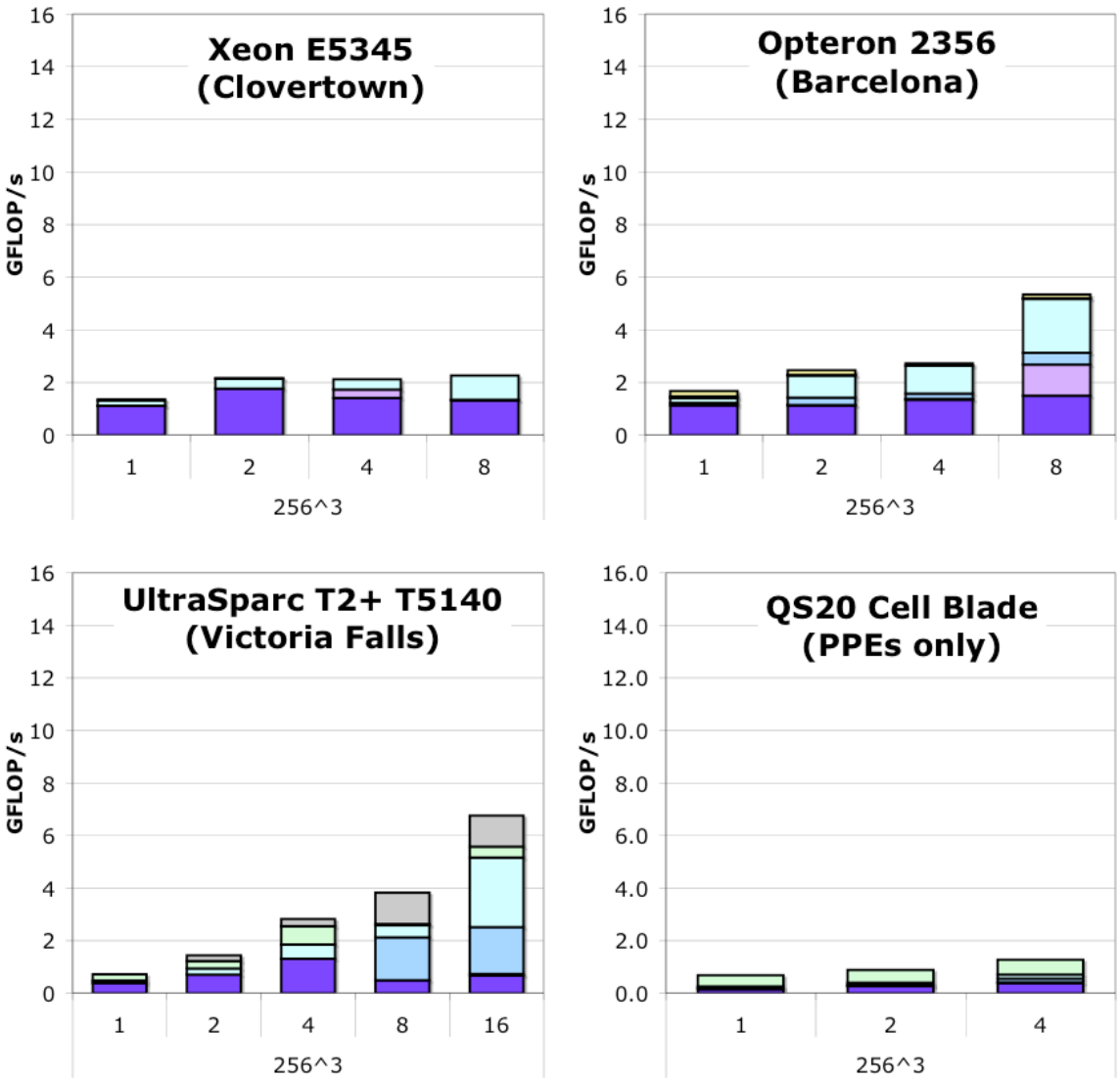
- Future Work:
  - cache bypass (e.g. *movntpd*)
  - software prefetching
  - SIMD intrinsics
  - data structure transformations

Stencil Performance
(out-of-the-box code)

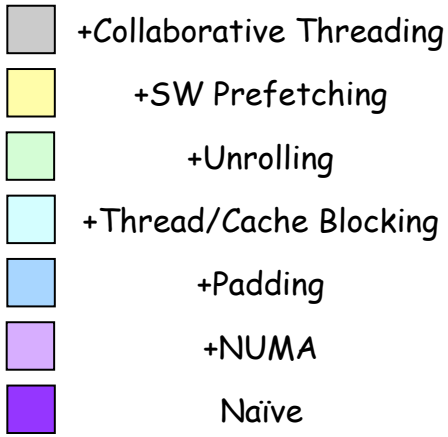


- Expect performance to be between SpMV and LBMHD
- Scalability is universally poor
- Performance is poor

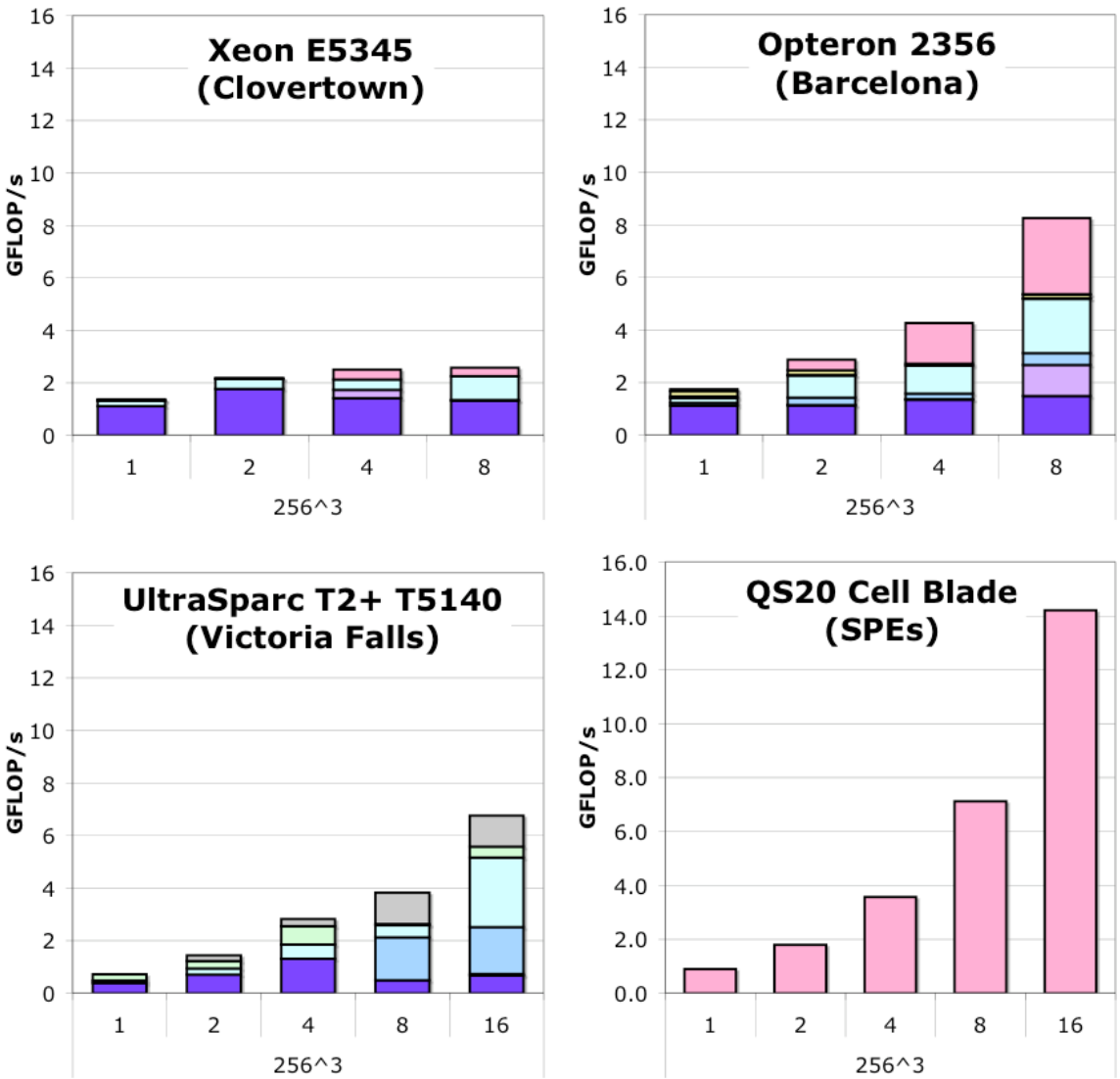
Auto-tuned Stencil Performance
(portable C)



- NUMA management is essential on most architectures
- Cache blocking is still essential even with MB's of cache



Auto-tuned Stencil Performance
(architecture specific optimizations)



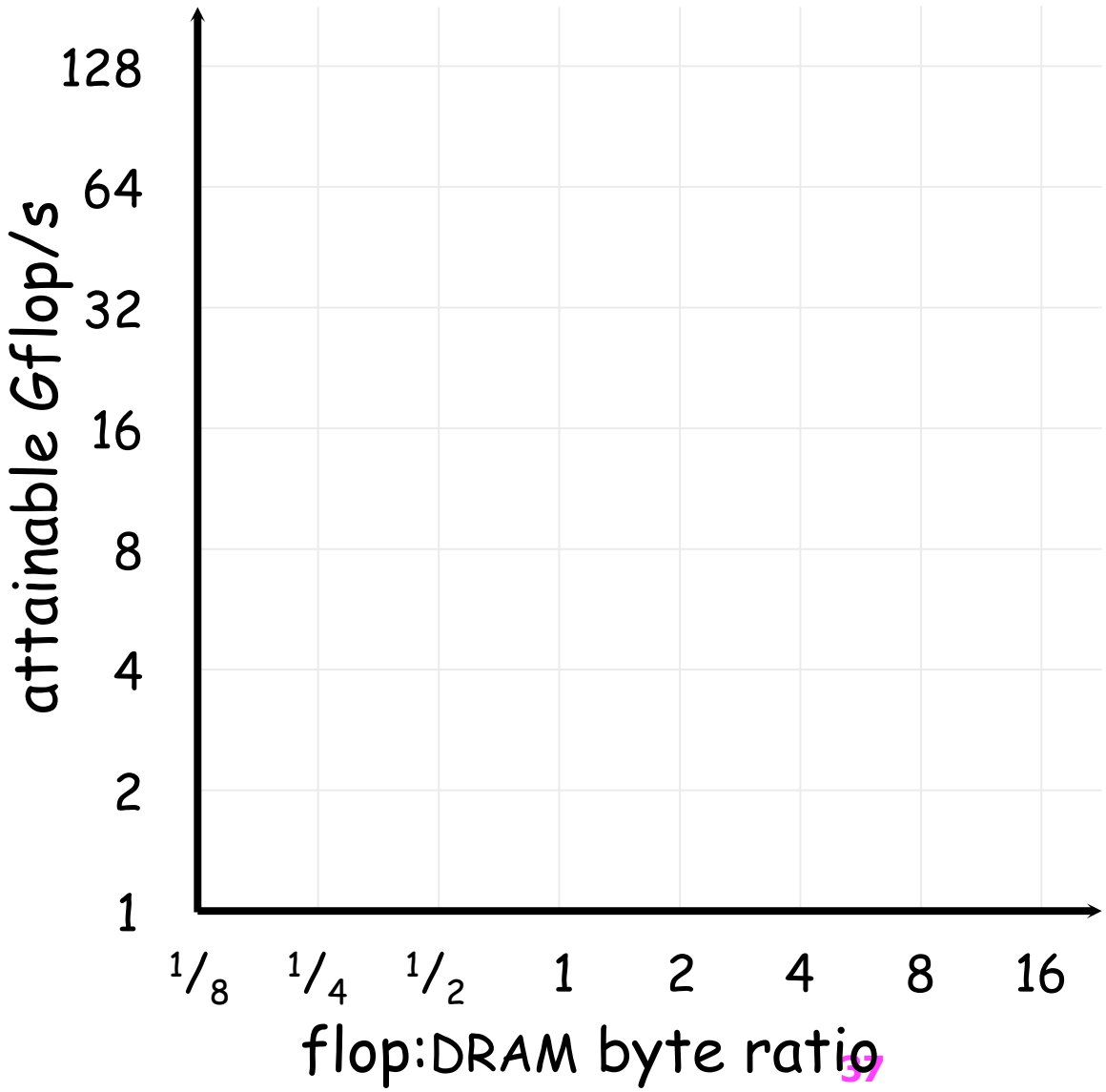
- Cache bypass can significantly improve Barcelona performance.
- DMA, SIMD, and cache blocking were essential on Cell

+Cache bypass / DMA
+Explicit SIMDization
+Collaborative Threading
+SW Prefetching
+Unrolling
+Thread/Cache Blocking
+Padding
+NUMA
Naïve

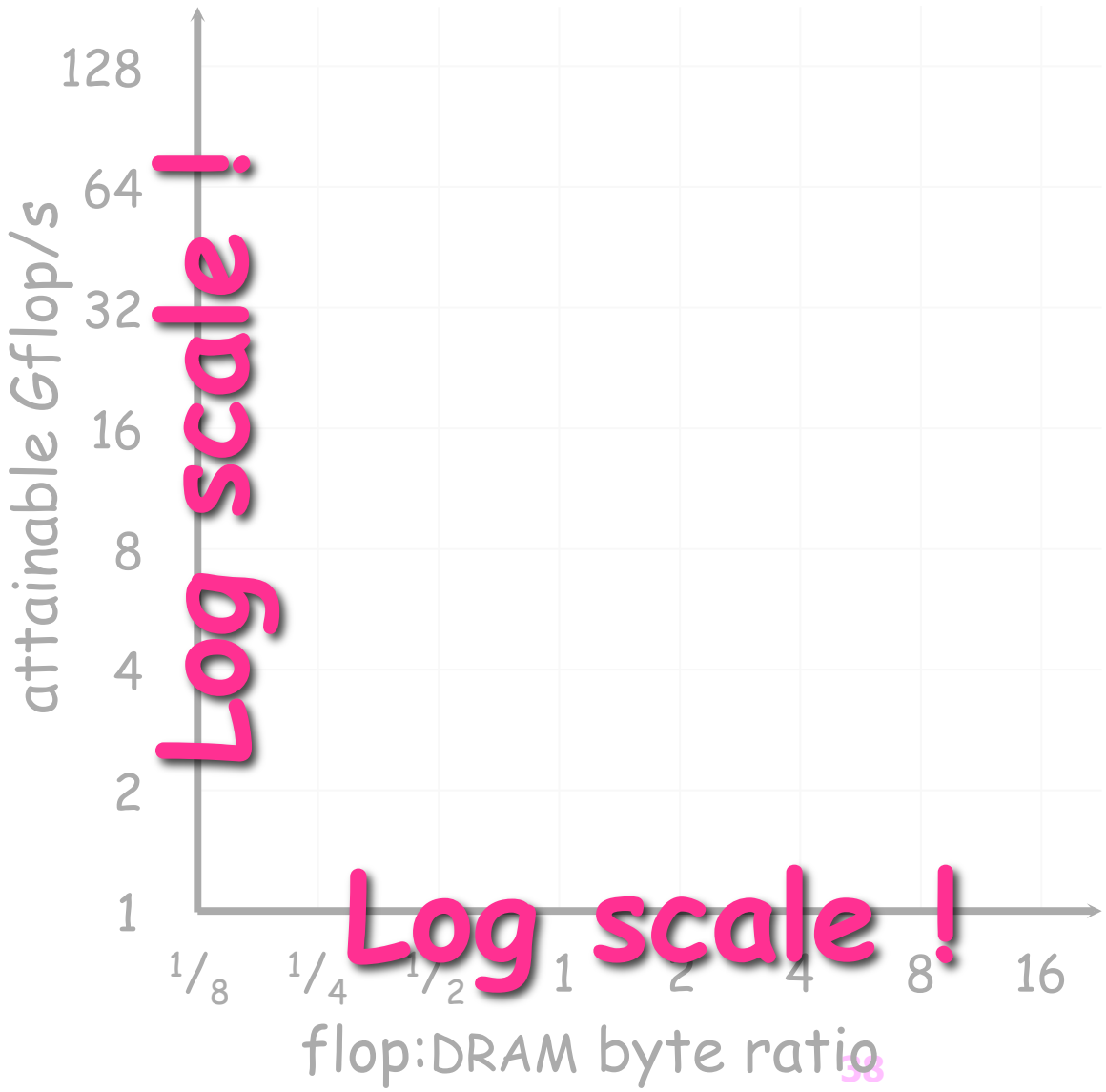
When to Say "Enough"

- Tuning significantly improved performance
- But is it good enough? What can we expect?
- Performance models are important for tuning
  - Understand your bottlenecks!
  - Performance pitfall: running out of memory bandwidth

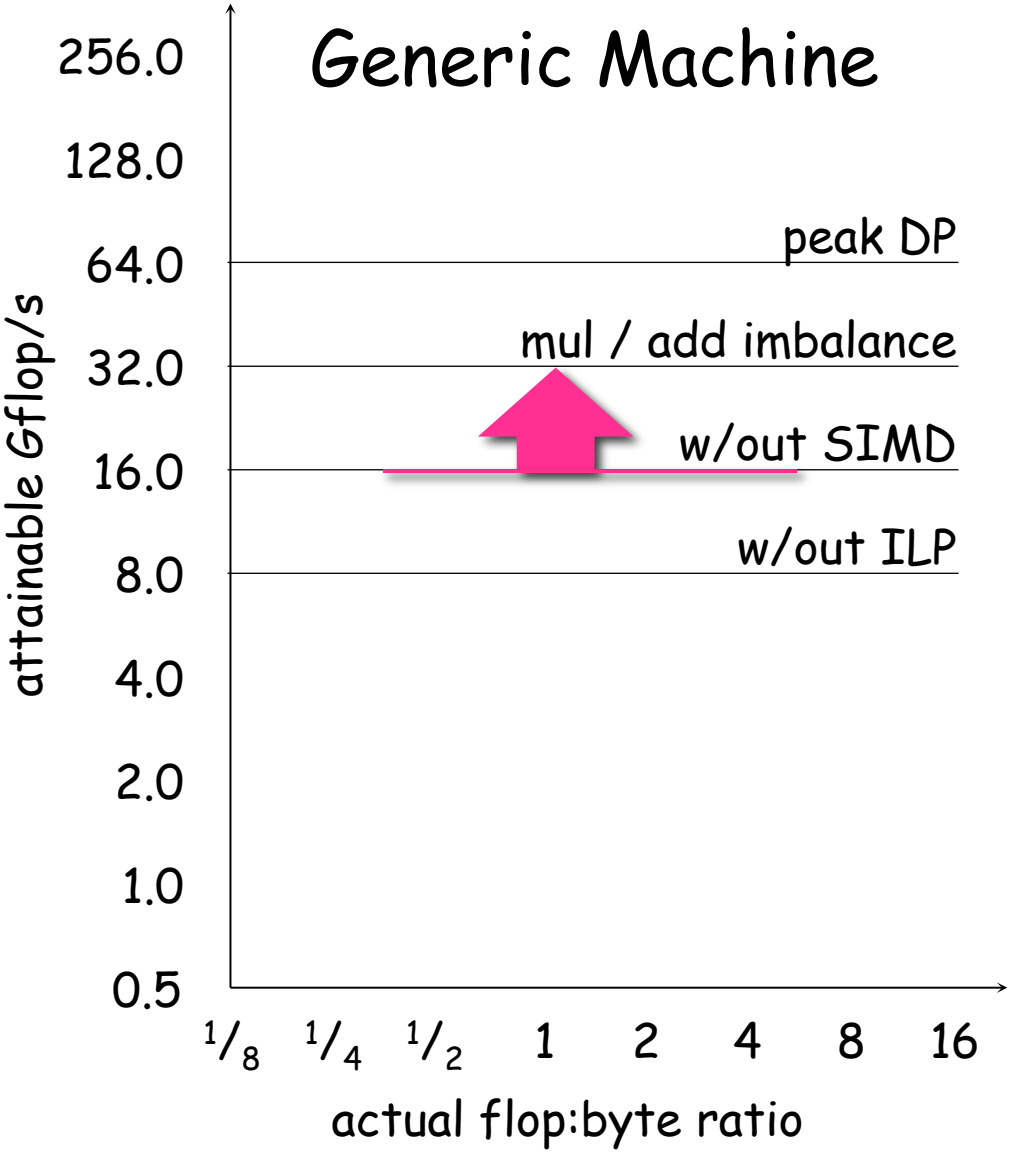
The Roofline Performance Model



The Roofline Performance Model

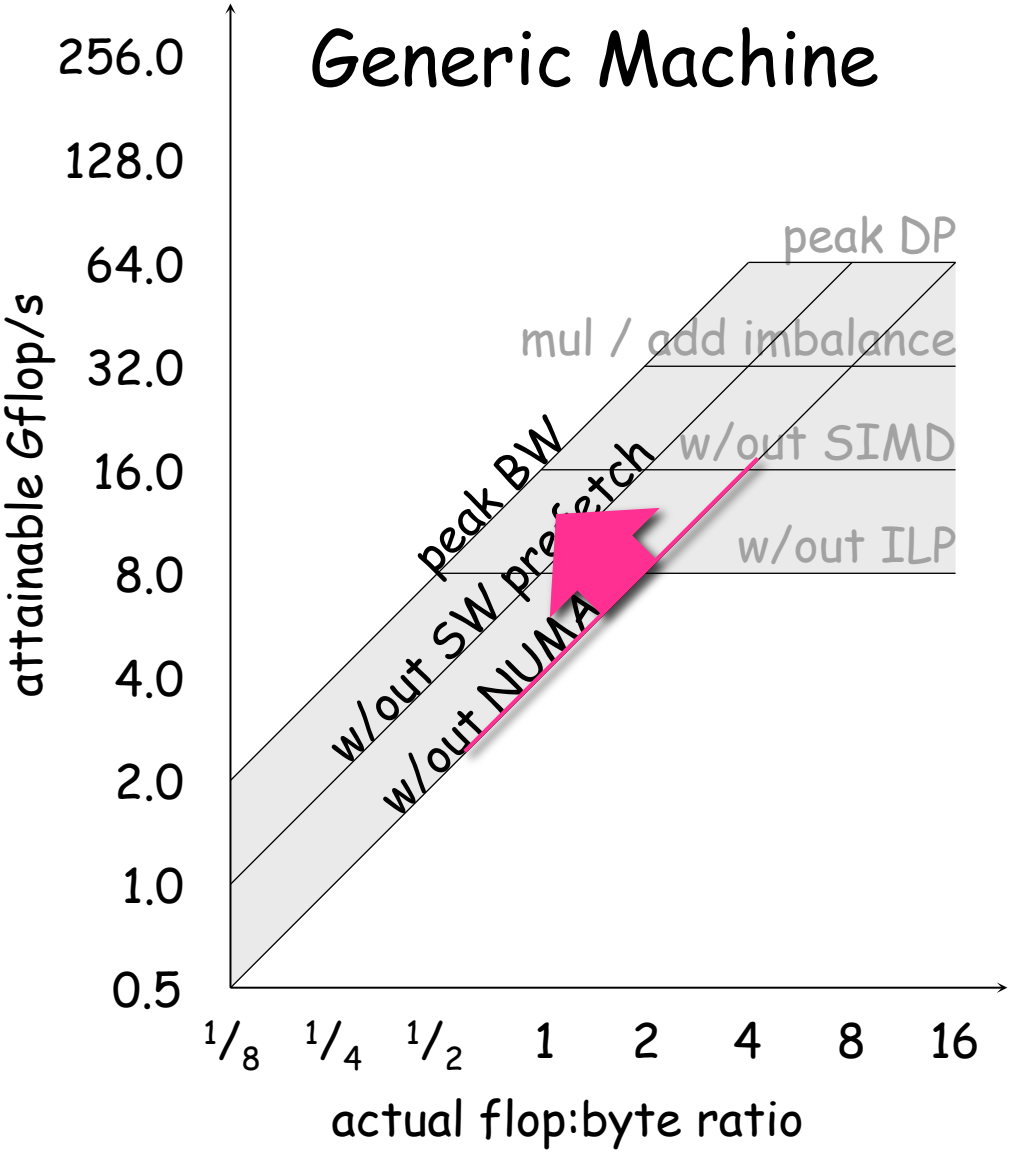


The Roofline Performance Model



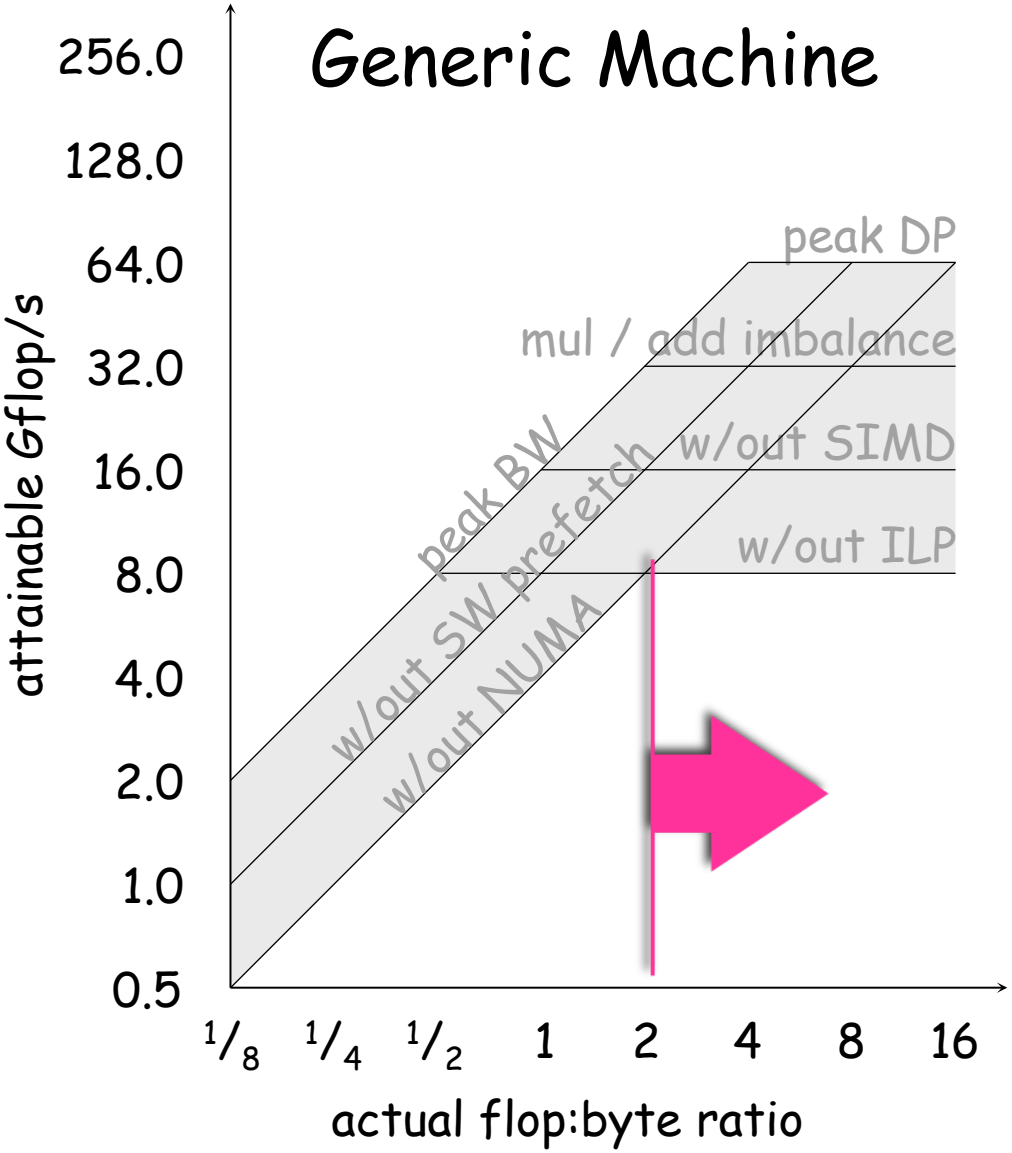
- The top of the roof is determined by peak computation rate (Double Precision floating point, DP for these algorithms)
- The instruction mix, lack of SIMD operations, ILP or failure to use other features of peak will lower attainable

The Roofline Performance Model



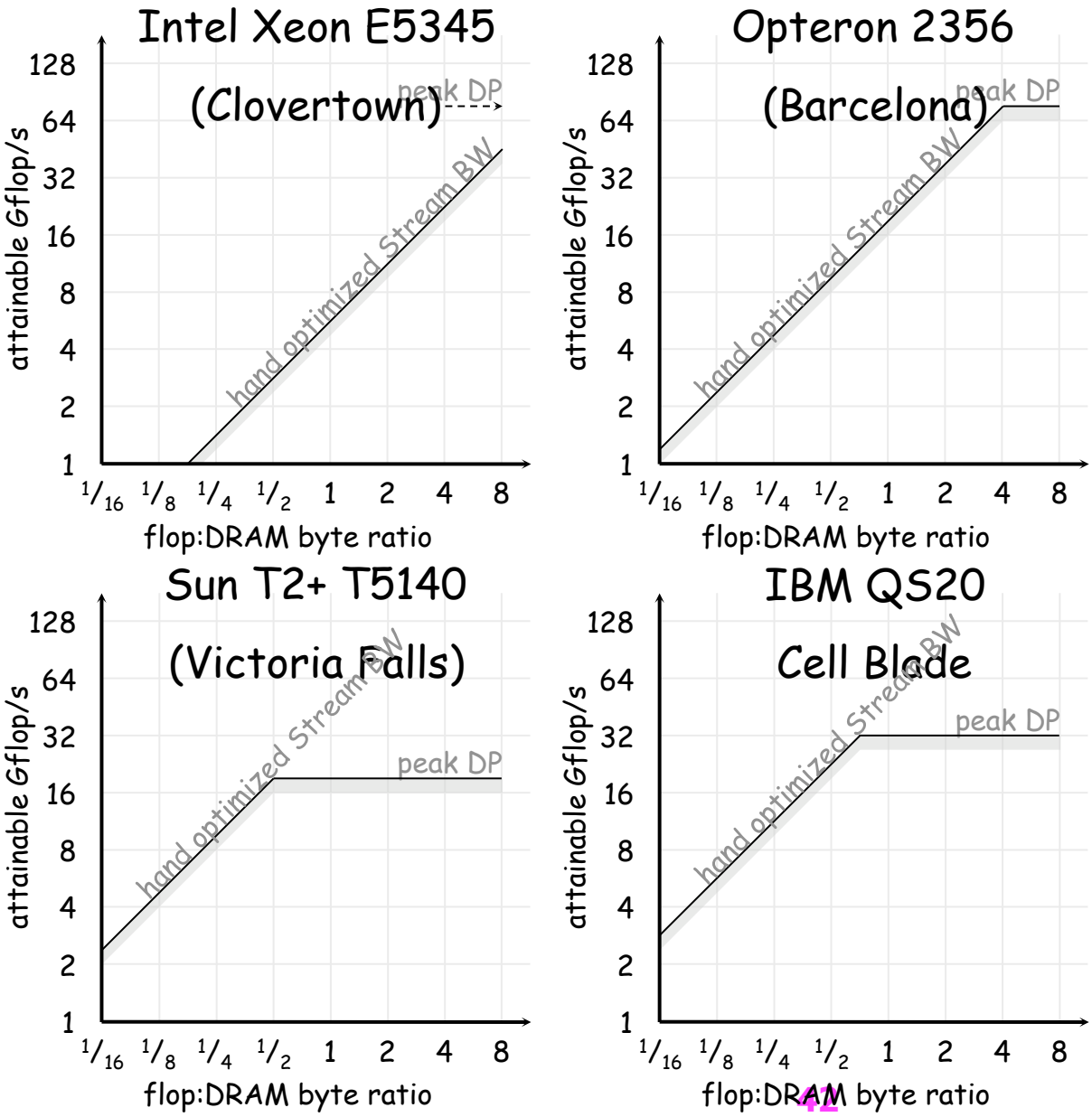
- The sloped part of the roof is determined by peak DRAM bandwidth (STREAM)
- Lack of proper prefetch, ignoring NUMA, or other things will reduce attainable bandwidth

The Roofline Performance Model



- Locations of posts in the building are determined by algorithmic intensity
  - Will vary across algorithms and with bandwidth-reducing optimizations, such as better cache re-use (tiling), compression techniques

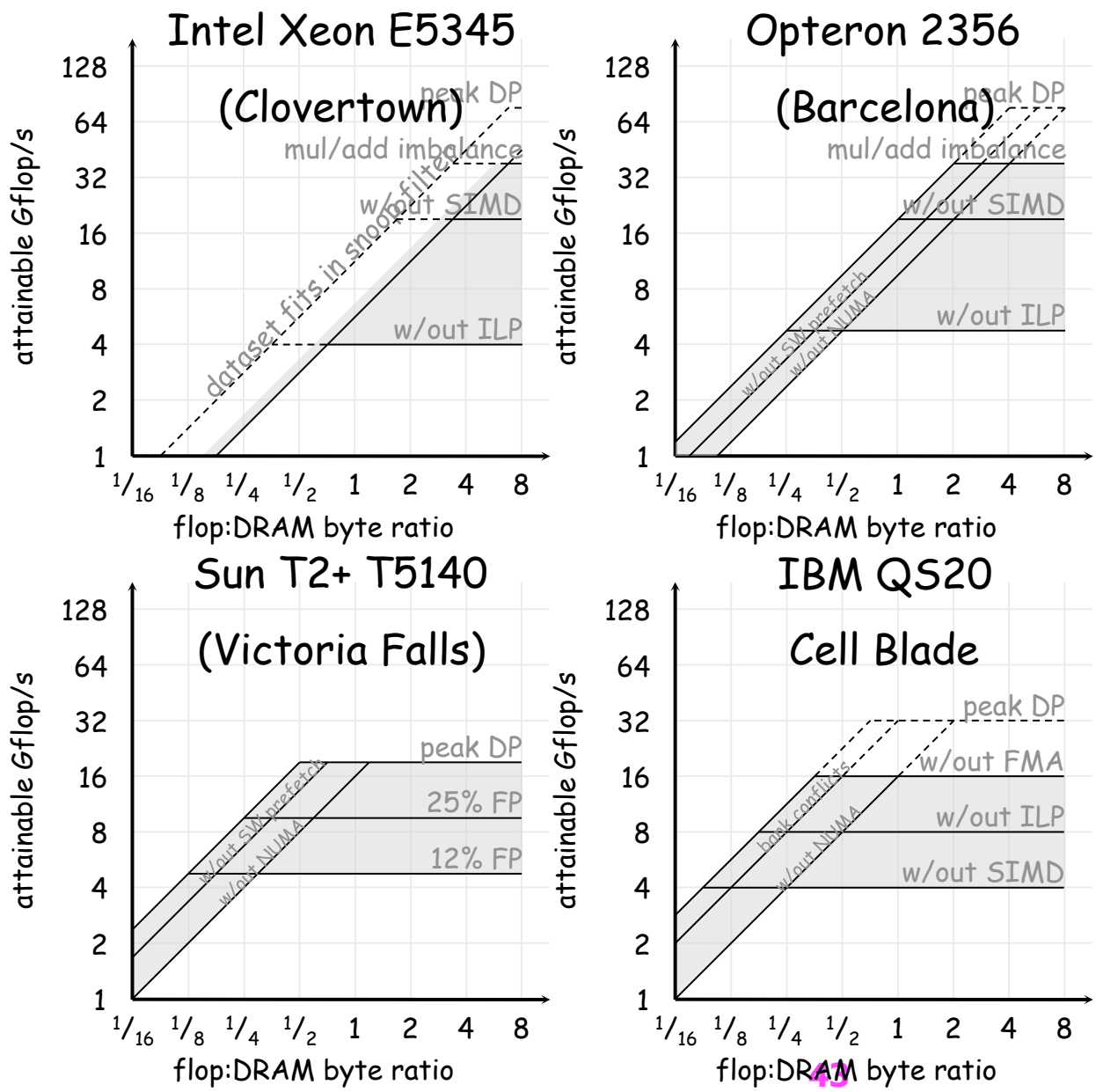
Naïve Roofline Model



- Unrealistically optimistic model
- Hand optimized Stream BW benchmark

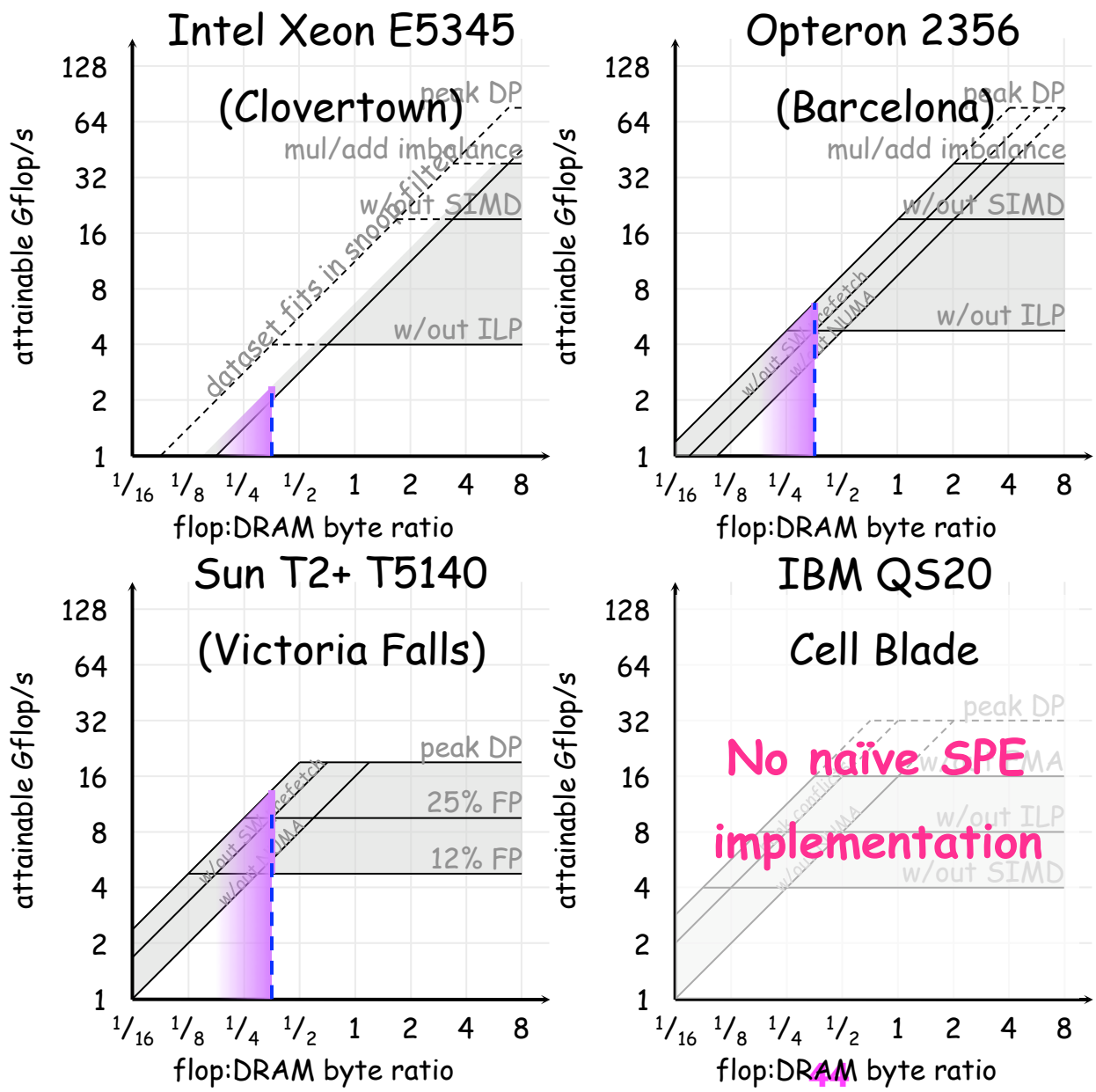
Gflop/s(AI) = min { Peak Gflop/s, StreamBW * AI }

Roofline model for Stencil
(out-of-the-box code)



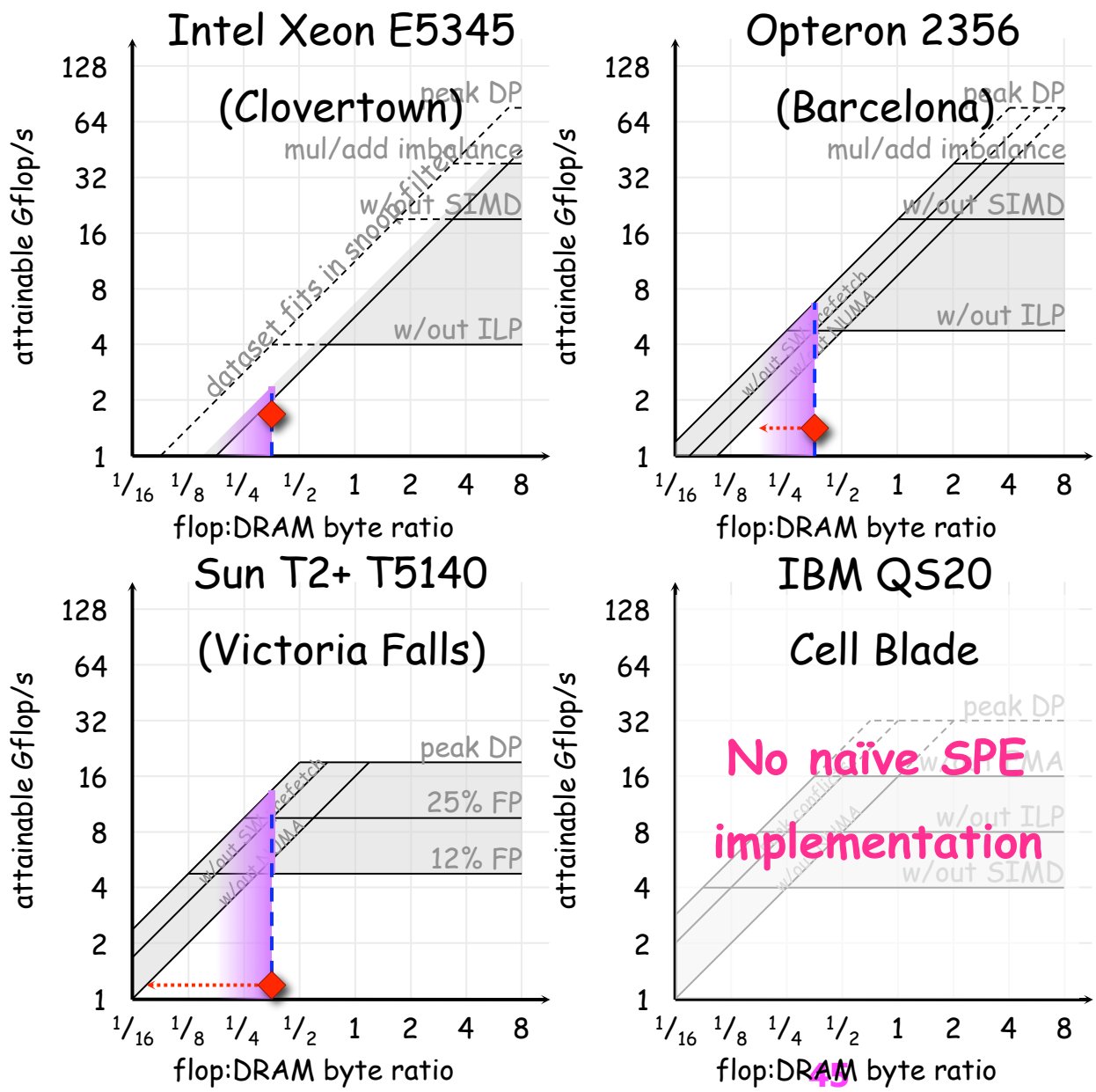
- Large datasets
  - 2 unit stride streams
  - No NUMA
  - Little ILP
  - No DLP
- Far more adds than multiplies (imbalance)
- Ideal flop:byte ratio $^1/_3$
  - High locality requirements
  - Capacity and conflict misses will severely impair flop:byte ratio

Roofline model for Stencil
(out-of-the-box code)



- Large datasets
  - 2 unit stride streams
  - No NUMA
  - Little ILP
  - No DLP
- Far more adds than multiplies (imbalance)
- Ideal flop:byte ratio $^1/_3$
  - High locality requirements
  - Capacity and conflict misses will severely impair flop:byte ratio

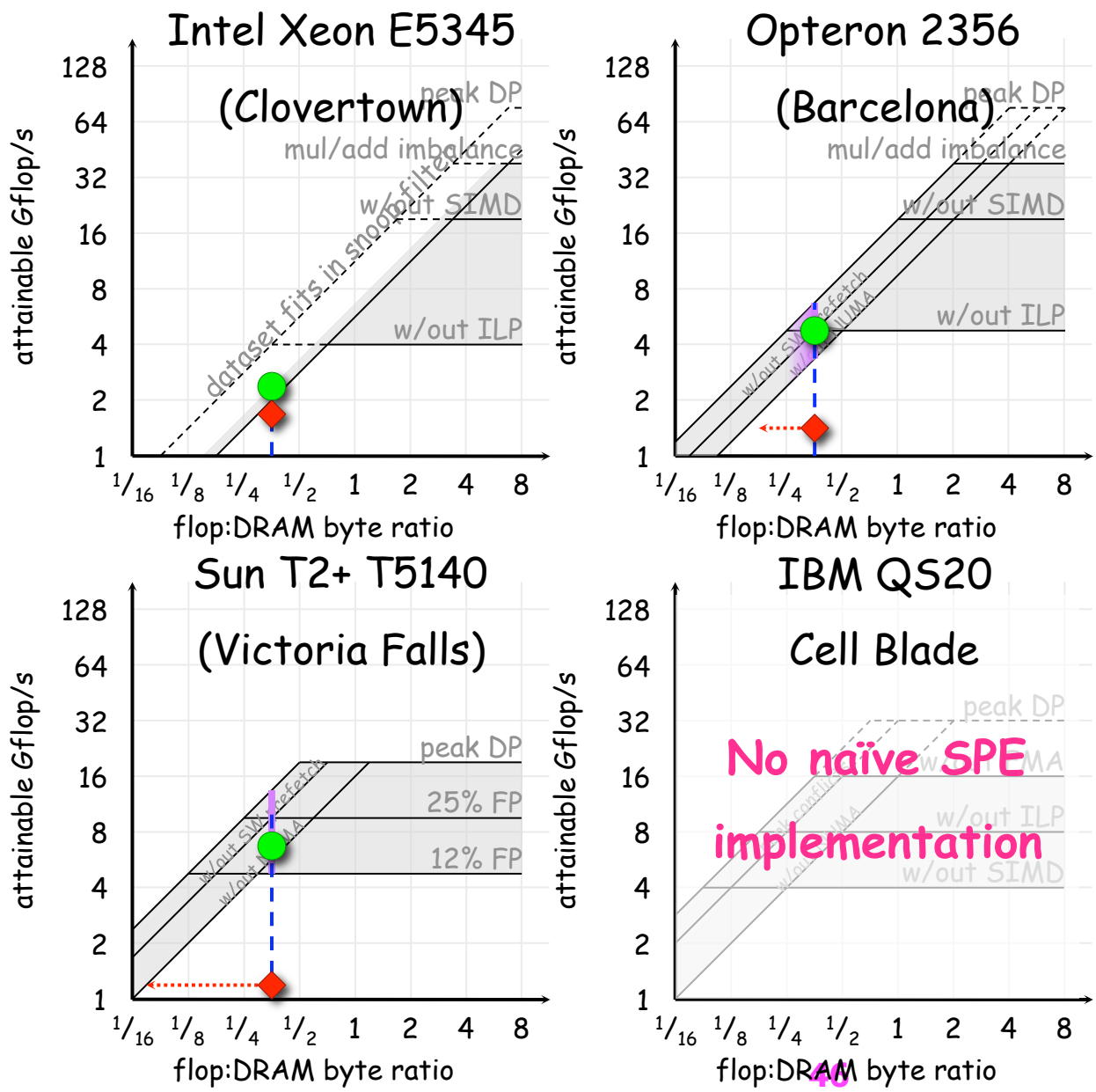
Roofline model for Stencil
(out-of-the-box code)



- Large datasets
  - 2 unit stride streams
  - No NUMA
  - Little ILP
  - No DLP
- Far more adds than multiplies (imbalance)
- Ideal flop:byte ratio $^1/_3$
  - High locality requirements
  - Capacity and conflict misses will severely impair flop:byte ratio

Roofline model for Stencil

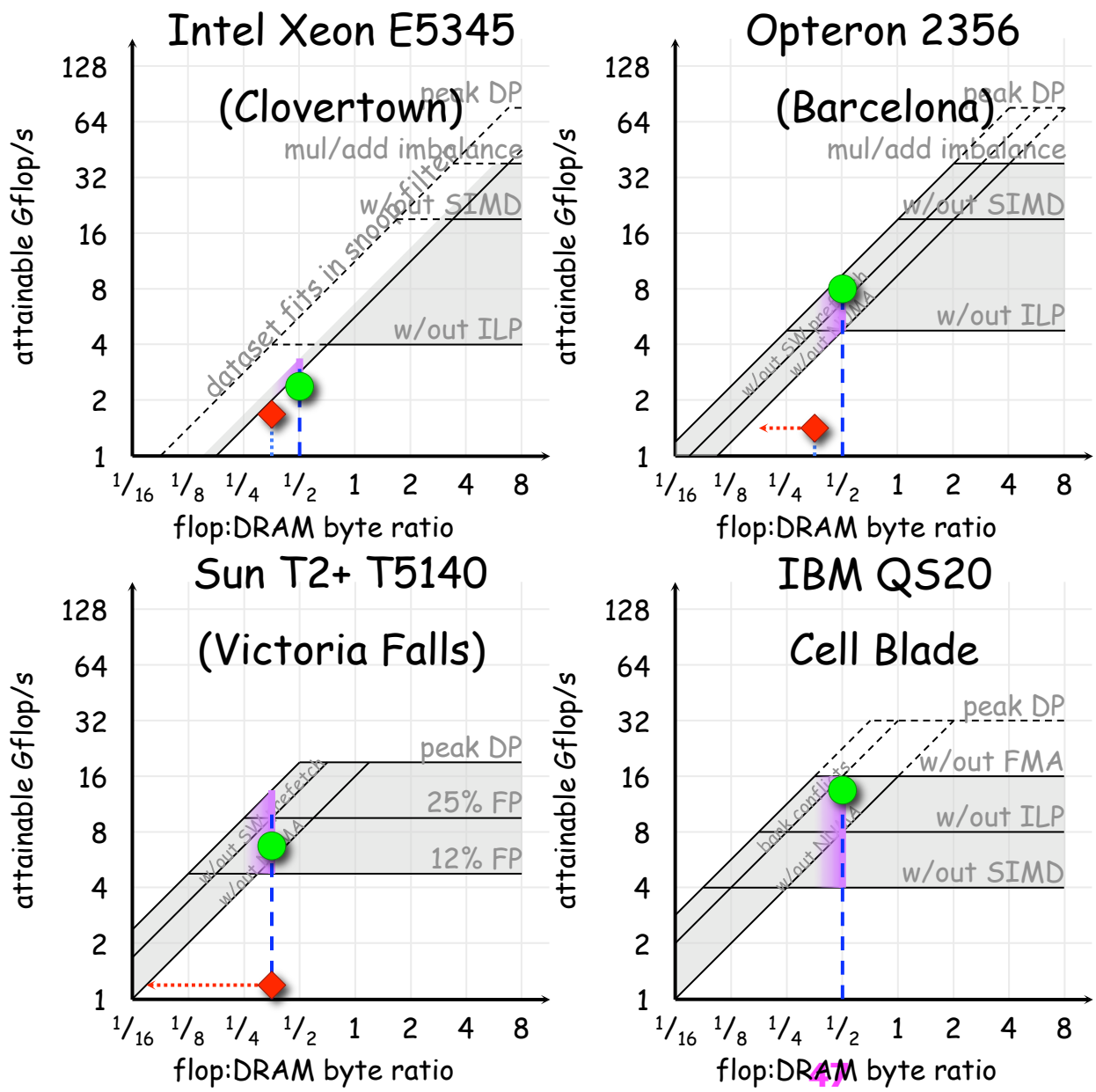
(NUMA, cache blocking, unrolling, prefetch, …)



- Cache blocking helps ensure flop:byte ratio is as close as possible to $^1/_3$
- Clovertown has huge caches but is pinned to lower BW ceiling
- Cache management is essential when capacity/thread is low

Roofline model for Stencil

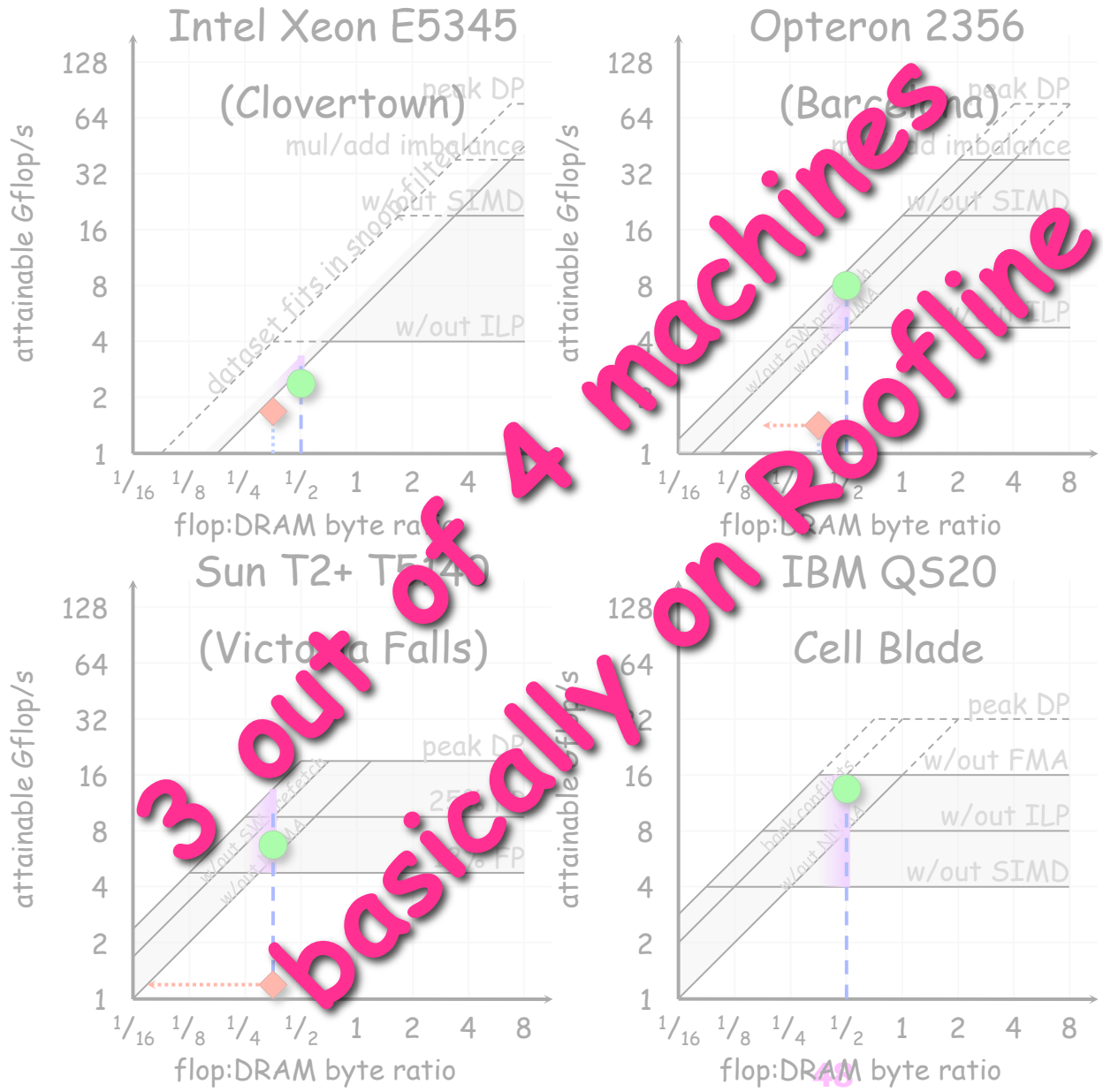
(SIMDization + cache bypass)



- Make SIMDization explicit
- Use cache bypass instruction: ***movntpd***
- Increases flop:byte ratio to ~0.5 on x86/Cell

Roofline model for Stencil

(SIMDization + cache bypass)



3 out of 4 machines basically on Roofline

- Make SIMDization explicit
  - Technically, this swaps ILP and SIMD ceilings
  - Use cache bypass instruction: ***movntpd***
- Increases flop:byte ratio to ~0.5 on x86/Cell

Conclusions

- Thread programming is not for the faint of heart
  - Can get close to the hardware
  - Use performance models to understand what that means
  - For more detailed tutorial info, see:
    https://computing.llnl.gov/tutorials/pthreads/
- Correctness problems abound
  - Data races
  - Deadlocks
- Performance problems include
  - Unthrottled parallelism
  - Oversynchronizing (locks or barriers)
  - Affinity control
  - Memory hierarchy
- But: this is the programming model of choice for people trying to get the best performance