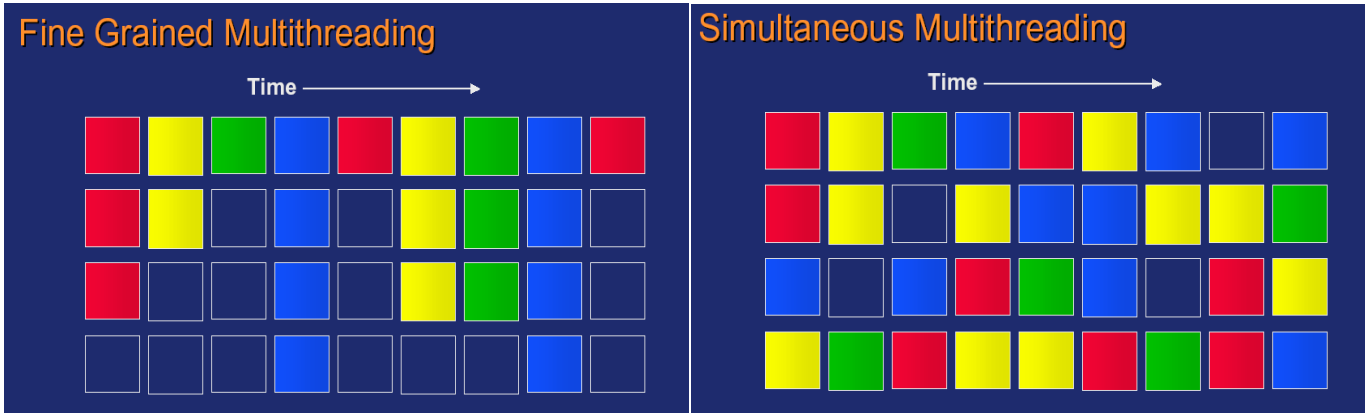
Advanced Computer Architecture Chapter 7:

Multi-threading

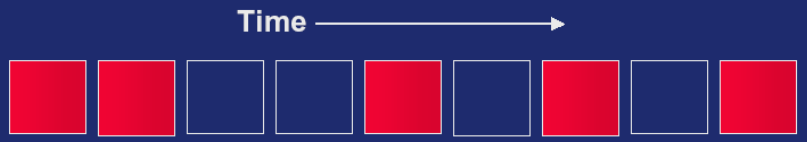


November 2026

Paul H J Kelly

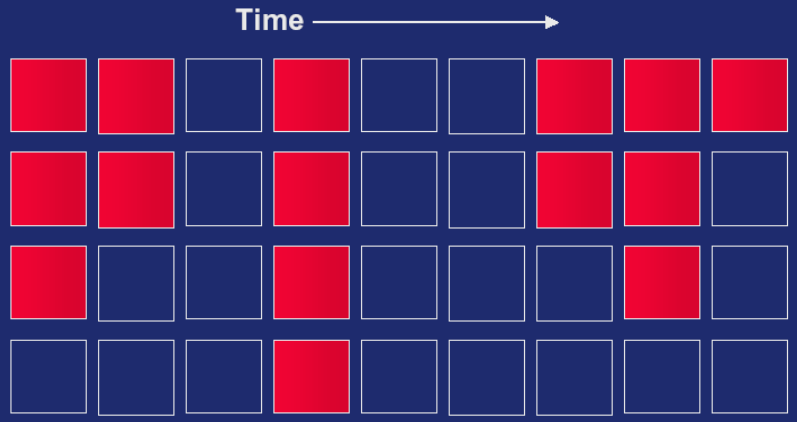
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd and 4th eds)*, and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

Instruction Issue



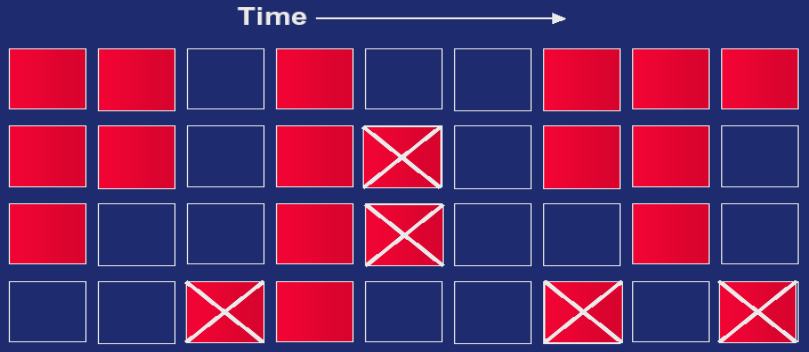
Reduced function unit utilization due to dependencies

Superscalar Issue



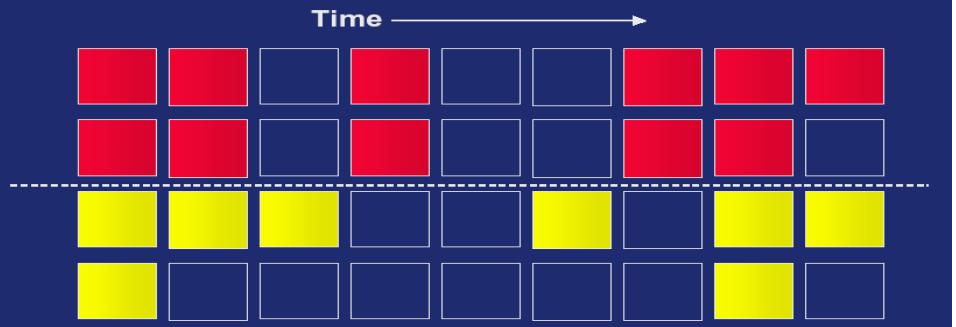
Superscalar leads to more performance, but lower utilization

Predicated Issue



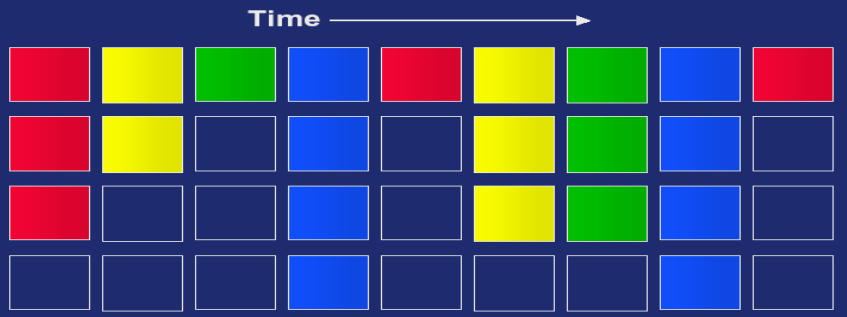
Adds to function unit utilization, but results are thrown away

Chip Multiprocessor



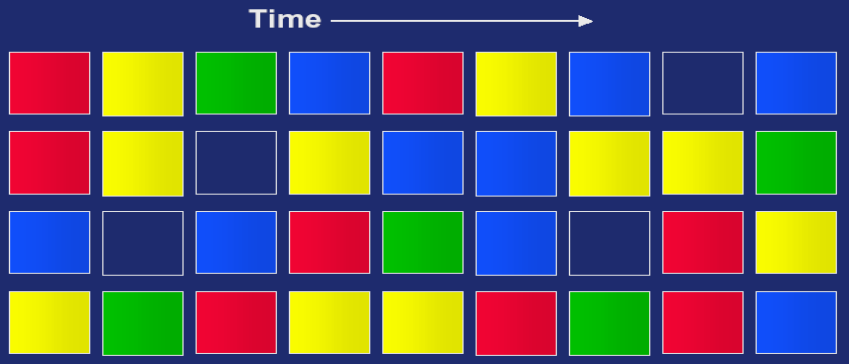
Limited utilization when only running one thread

Fine Grained Multithreading



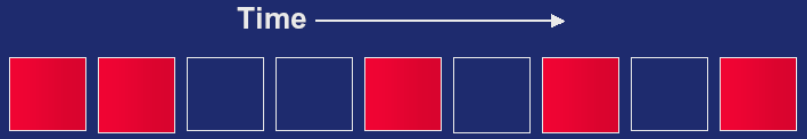
Intra-thread dependencies still limit performance

Simultaneous Multithreading



Maximum utilization of function units by independent operations

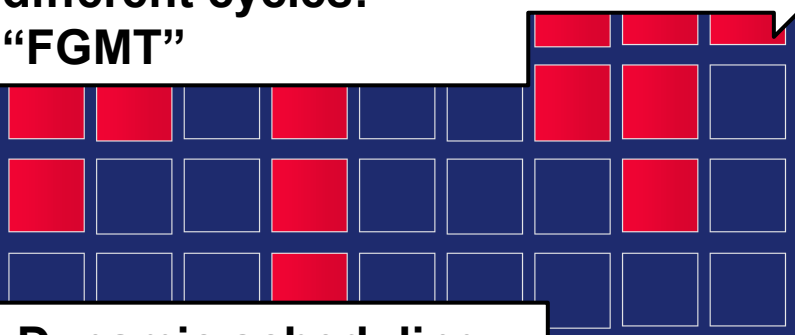
Instruction Issue



Reduced function unit utilization due to dependencies

Superscalar Issue

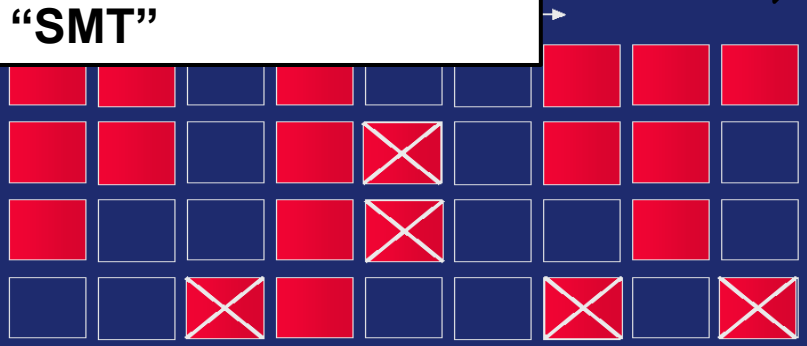
Different threads in different cycles: "FGMT"



ut lower utilization

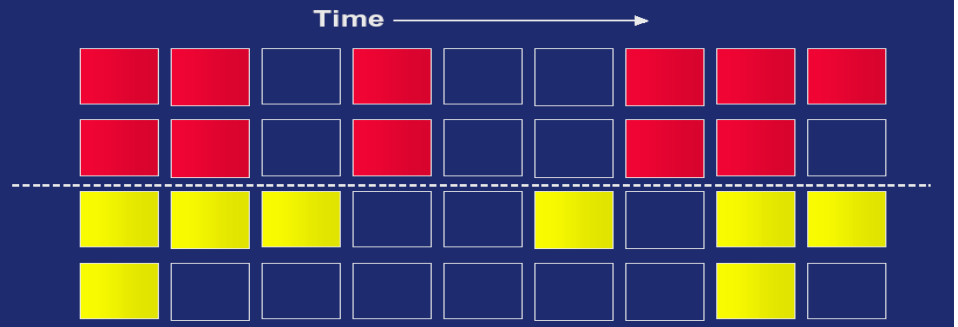
Pre

Dynamic scheduling of operations from a pool of threads: "SMT"



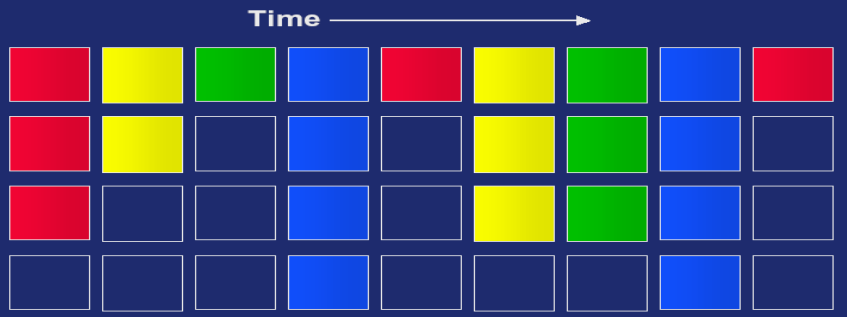
Adds to function unit utilization, but results are thrown away

Chip Multiprocessor



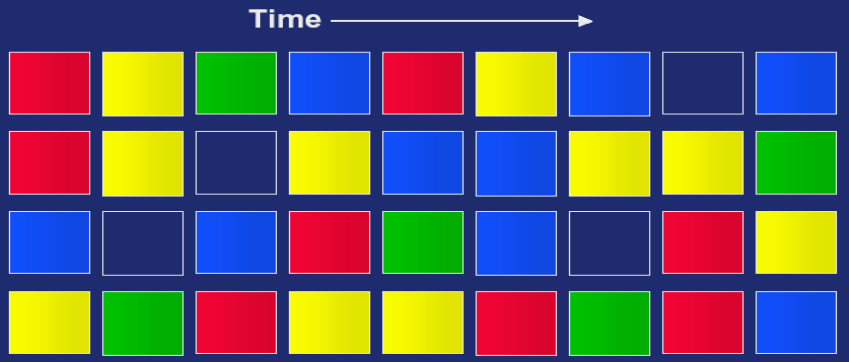
Limited utilization when only running one thread

Fine Grained Multithreading



Intra-thread dependencies still limit performance

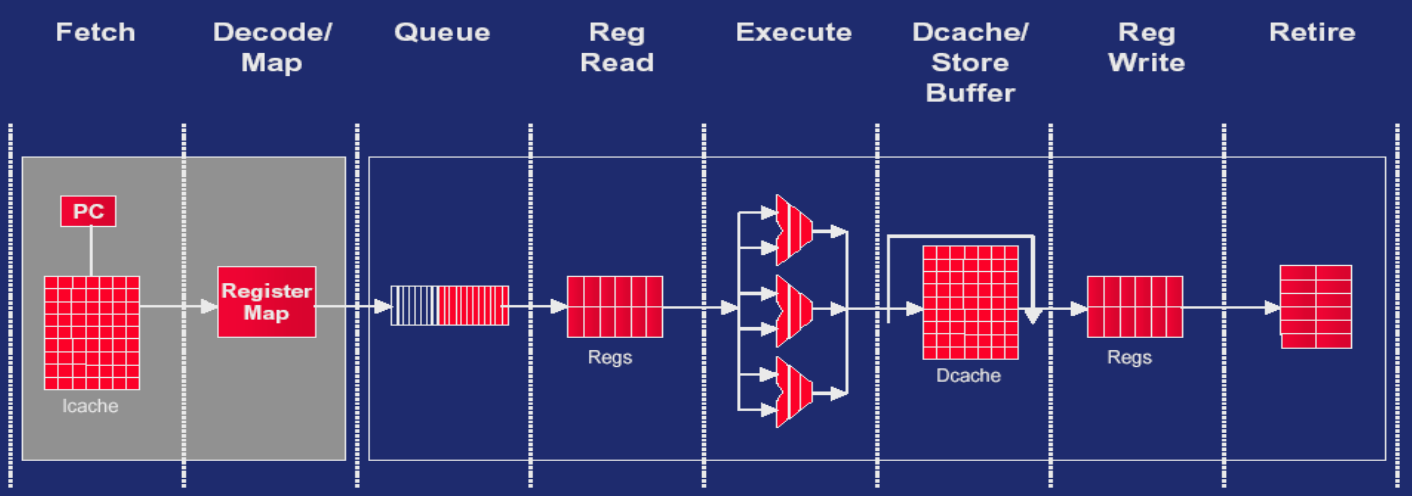
Simultaneous Multithreading



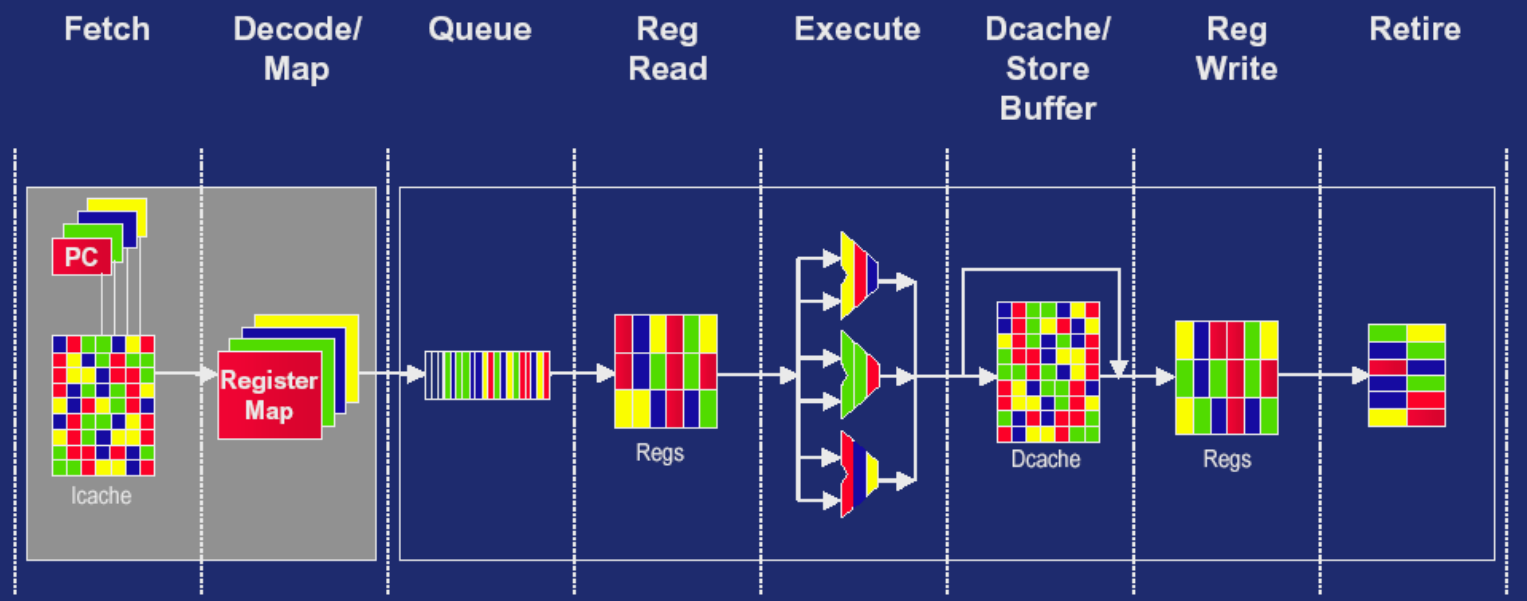
Maximum utilization of function units by independent operations

SMT

Basic Out-of-order Pipeline

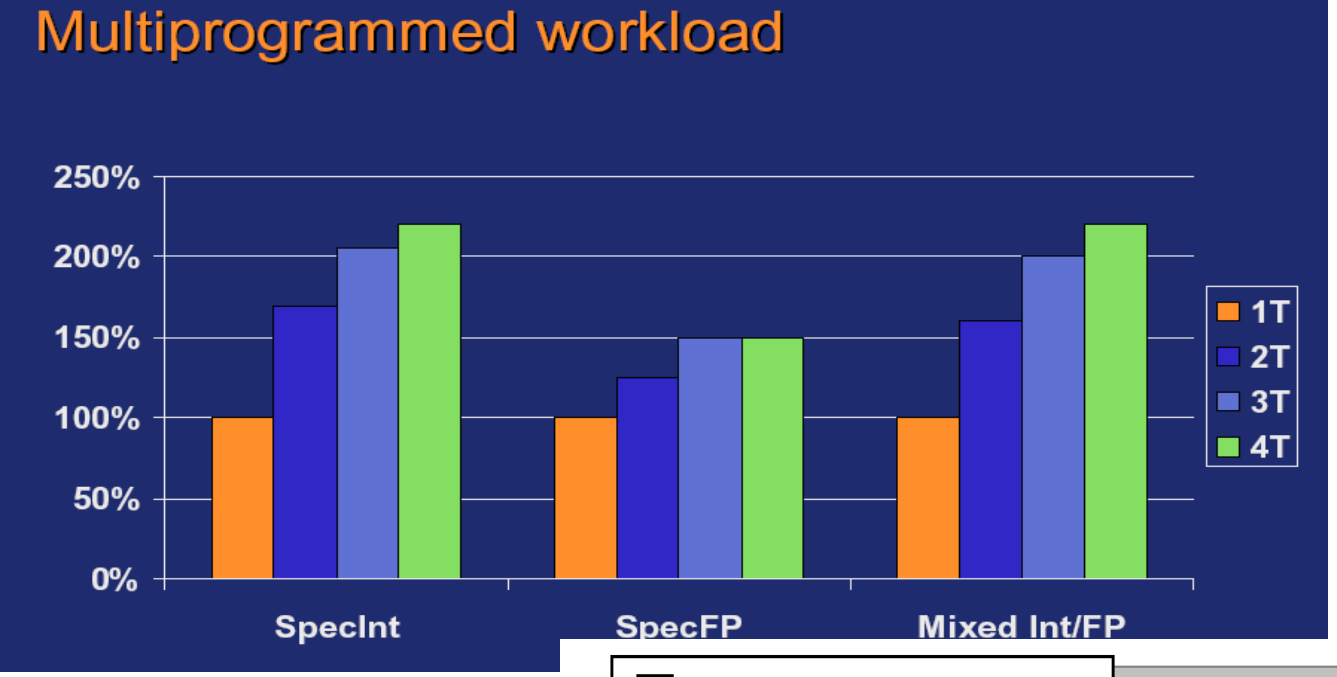


SMT Pipeline



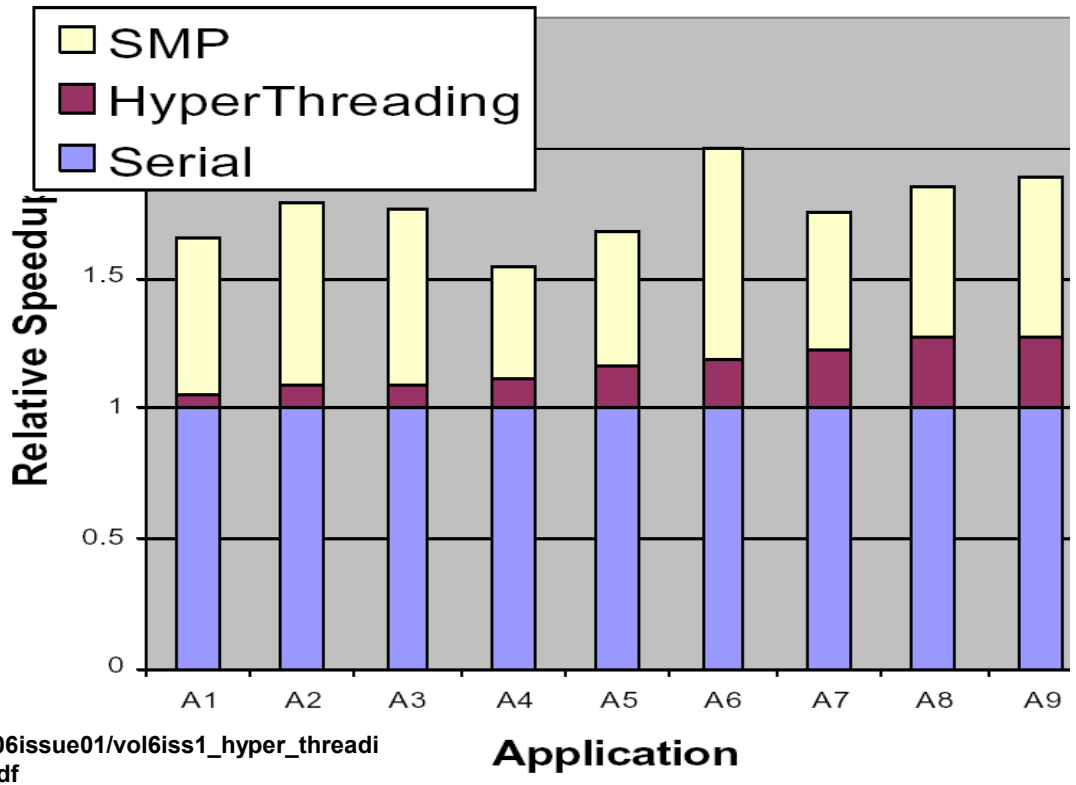
- Alpha 21464
- One CPU with 4 Thread Processing Units (TPUs)
- “6% area overhead over single-thread 4-issue CPU”

SMT performance



- Alpha 21464

- Intel Pentium 4 with hyperthreading:



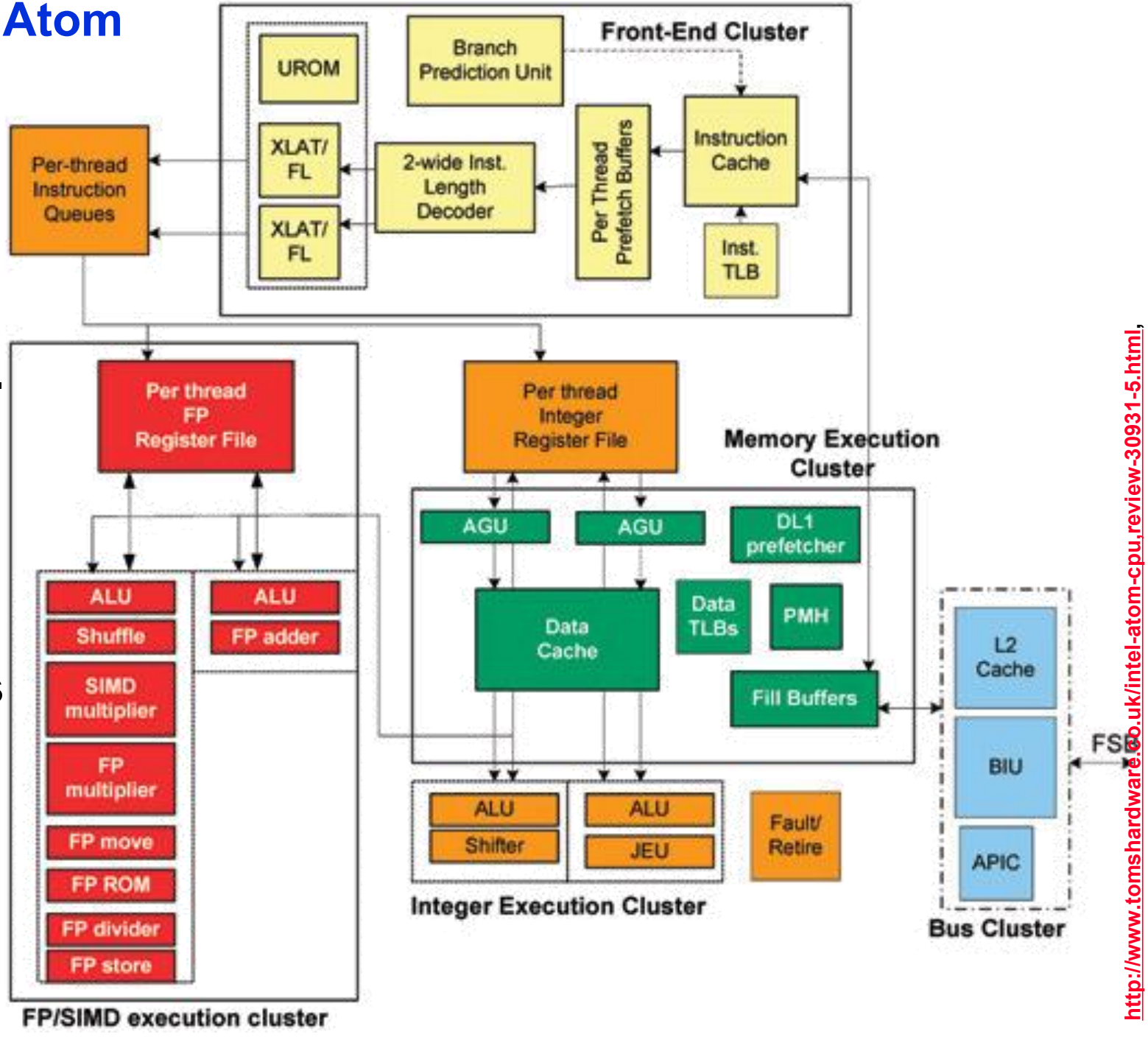
Code	Description
A1	Mechanical Design Analysis (finite element method) This application is used for metal-forming, drop testing, and crash simulation.
A2	Genetics A genetics application that correlates DNA samples from multiple animals to better understand congenital diseases.
A3	Computational Chemistry This application uses the self-consistent field method to compute chemical properties of molecules such as new pharmaceuticals.
A4	Mechanical Design Analysis This application simulates the metal-stamping process.
A5	Mesoscale Weather Modeling This application simulates and predicts mesoscale and regional-scale atmospheric circulation.
A6	Genetics This application is designed to generate Expressed Sequence Tags (EST) clusters, which are used to locate important genes.
A7	Computational Fluid Dynamics This application is used to model free-surface and confined flows.
A8	Finite Element Analysis This finite element application is specifically targeted toward geophysical engineering applications.
A9	Finite Element Analysis This explicit time-stepping application is used for crash test studies and computational fluid dynamics.

http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

SMT in the Intel Atom (Silverthorne)

- Intel's bid to steal back some of the low-power market for IA-32 and Windows
- In-order
- 2-way SMT
- 2 instructions per cycle

(from same or different threads)



http://www.tomshardware.co.uk/intel-atom-cpu,review-30931-5.html,
https://arstechnica.com/gadgets/2008/02/small-wonder-inside-intels-silverthorne-ultramobile-cpu/

SMT issues

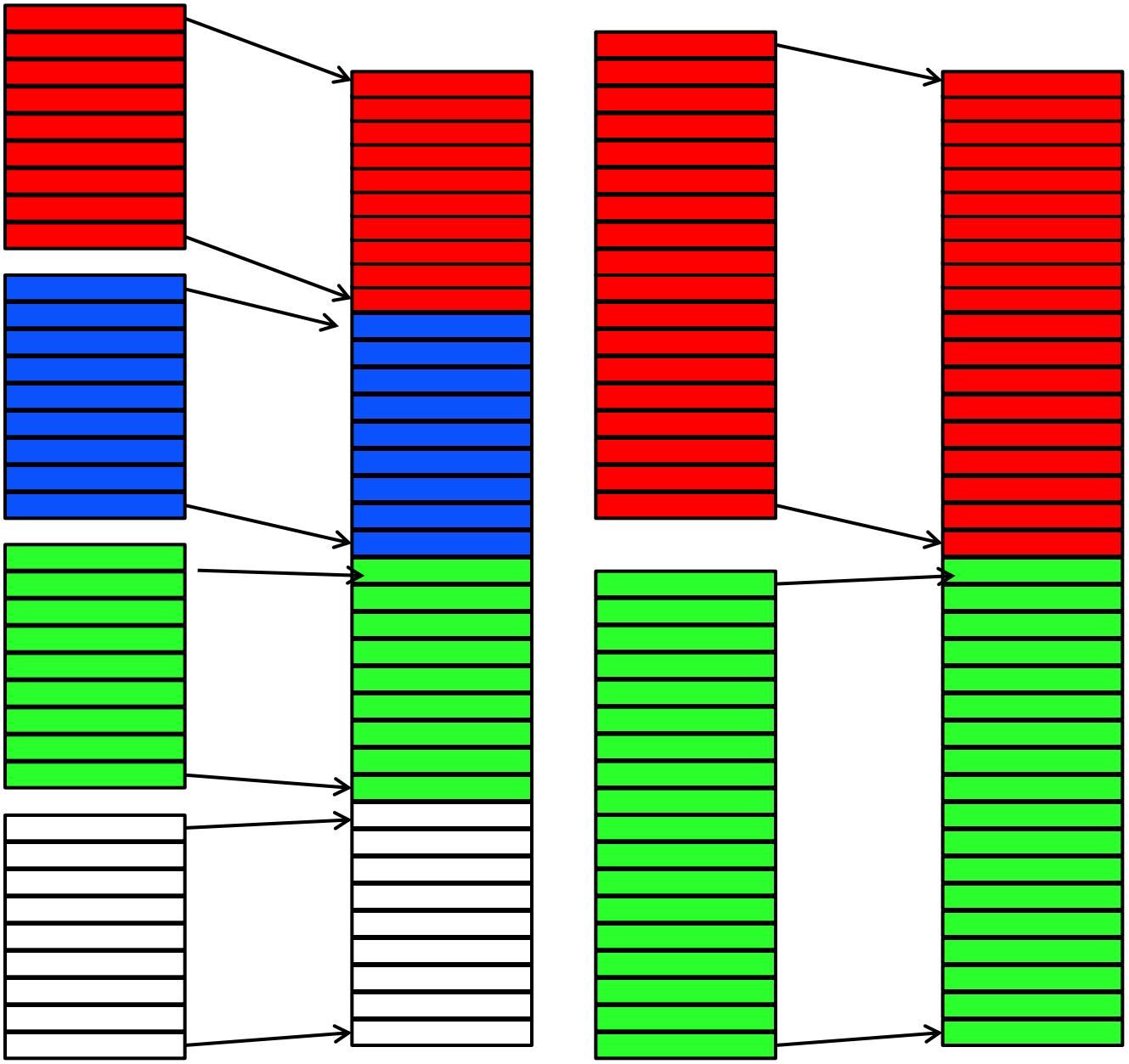
- Each thread runs slow?
  - The point of Simultaneous Multithreading is that resources are dynamically assigned, so if only one thread can run it can run faster
- SMT threads contend for resources
  - Possibly symbiotically?
    - One thread is memory-intensive, one arithmetic-intensive?
  - Possibly destructively
    - thrashing the cache? Other shared resources…. (TLB?)
- Which resources should be partitioned per-thread, and which should be shared on-demand?
- SMT threads need to be scheduled *fairly*
  - Can one thread monopolise the whole CPU?
    - Denial of service risk
    - Slow thread that suffers lots of cache misses fills ROB and blocks issue
- Side channels:
  - one thread may be able observe another's traffic and deduce what it's doing

SMT – latency-hiding

- SMT threads exploit memory-system parallelism
  - Easy way to get lots of memory accesses in-flight
  - "Latency hiding" – overlapping data access with compute
- What limits the number of threads we can have?
- SMT threads need a *lot* of registers
  - A lot of logical registers – but they share physical registers?

- In a machine *without* register renaming
  - What about *statically* partitioning the register file based on the number of registers each thread actually needs?
  - This is what many GPUs do
  - Leads to tradeoff: lots of lightweight threads to maximise latency hiding? Or fewer heavyweight threads that benefit from lots of registers?
  - Nvidia and AMD call this "occupancy"

Mapping threads into the register file



- If each thread needs few registers, we can have lots of them co-existing in the same physical register file
- Alternatively, we could have fewer, fatter threads
- More threads=higher “occupancy”
- Better latency hiding
- Tricky tradeoff!

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11.1	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	128	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

#registers per thread

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	33%

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	16	2
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6		
Limited by Registers per Multiprocessor	**2**	**8**	**16**
Limited by Shared Memory per Multiprocessor	32		

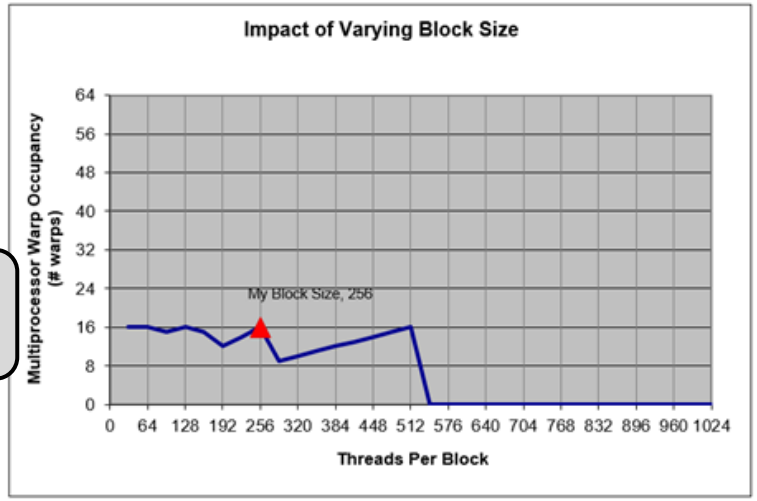
Note: Occupancy limiter is shown in orange

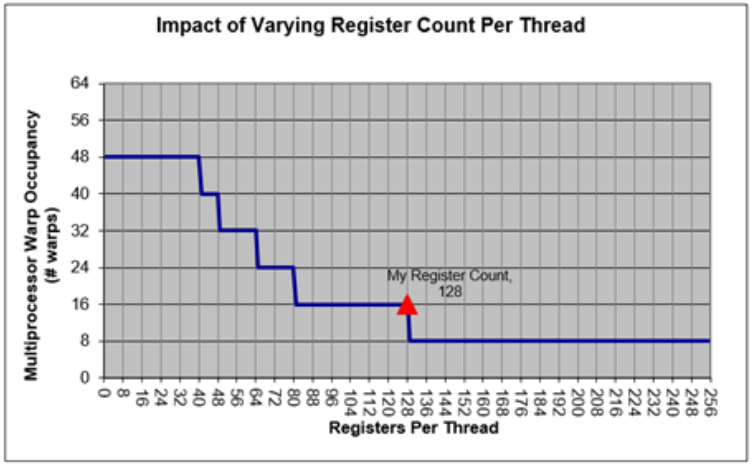
Physical Max Warps/SM = 48

Occupancy = 16 / 48 = 33%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11.1	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	64	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	67%

#registers per thread

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	32	4
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6		
Limited by Registers per Multiprocessor	**4**	**8**	**32**
Limited by Shared Memory per Multiprocessor	32		

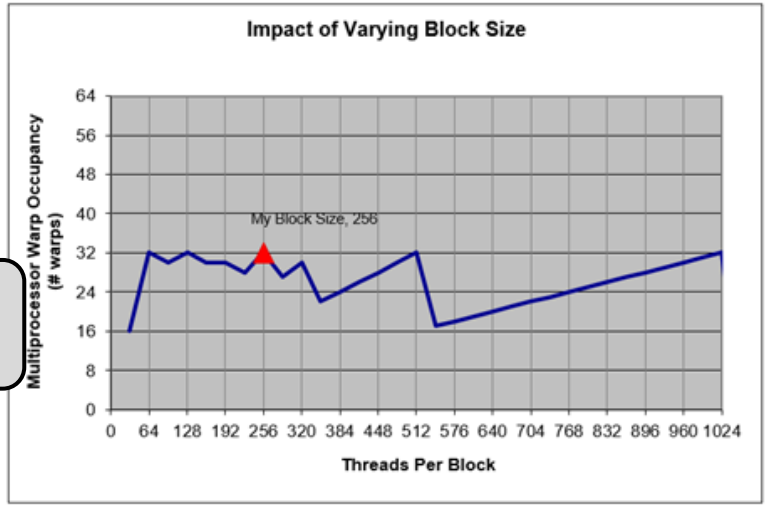
Note: Occupancy limiter is shown in orange

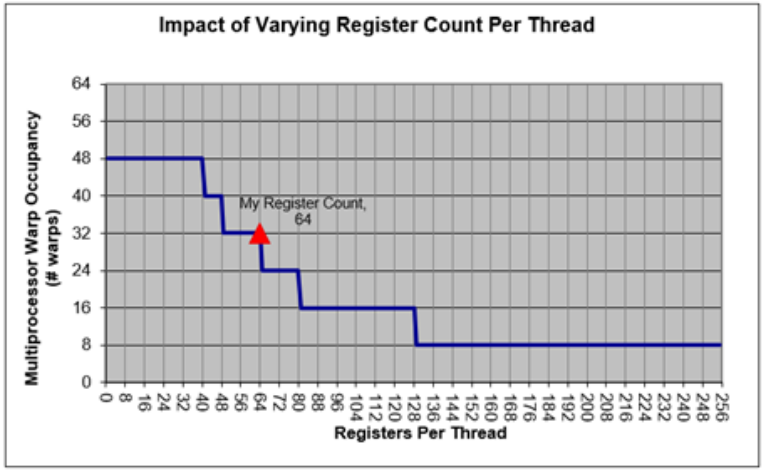
Physical Max Warps/SM = 48

Occupancy = 32 / 48 = 67%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11.1	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	6
Occupancy of each Multiprocessor	100%

#registers per thread

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48
Limited by Registers per Multiprocessor	8		
Limited by Shared Memory per Multiprocessor	32		

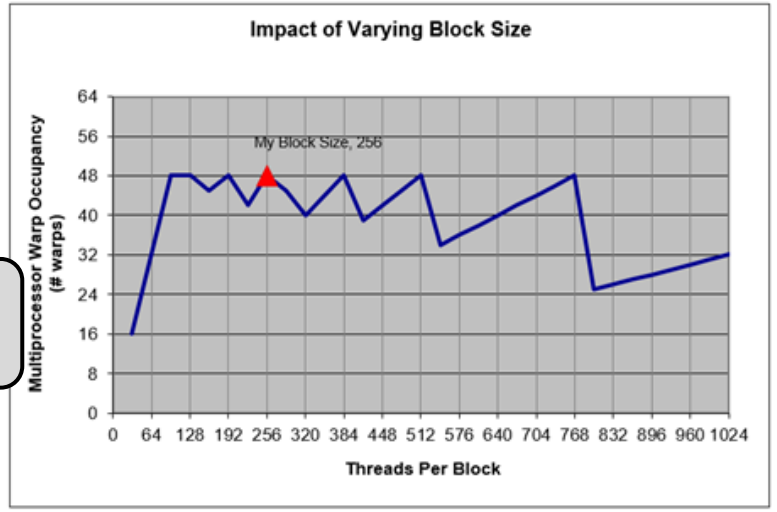
Note: Occupancy limiter is shown in orange

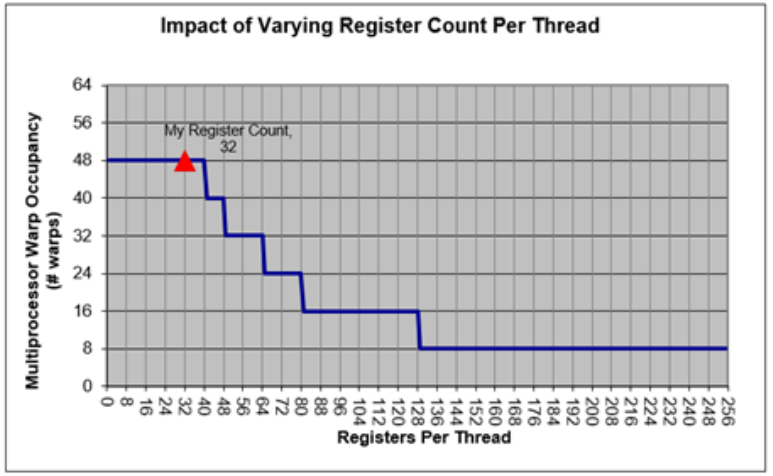
Physical Max Warps/SM = 48

Occupancy = 48 / 48 = 100%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





Chapter summary

- We have explored:
  - Pipeline parallelism
  - Dynamic instruction scheduling
  - Static instruction scheduling
  - Multiple instructions per cycle
  - Very long instruction words (VLIW)
  - Multi-threading
    - Coarse-grain
    - Fine-grain
    - Simultaneous multithreading (SMT)
    - Statically-partitioned multithreading

Vector instructions and SIMD – coming soon

SIMT and GPUs – coming soon

Multicore – coming soon

Extra slides for interest/fun

Is the "minimum" operator associative?

- **min(min(X, Y), Z = min(X, min(Y, Z)) ?**
- **min(X, Y) = if X<Y then X else Y**

min(min(10, x), 100) = 100

Extra slides for interest/fun

Is the "minimum" operator associative?

- **min(min(X, Y), Z = min(X, min(Y, Z)) ?**
- **min(X, Y) = if X<Y then X else Y**

All comparisons on NaNs always fail….

- **min(min(10, NaN), 100) = 100**

Extra slides for interest/fun

Is the "minimum" operator associative?

- **min(min(X, Y), Z = min(X, min(Y, Z)) ?**
- **min(X, Y) = if X<Y then X else Y**

All comparisons on NaNs always fail….

- **min(X , NaN) = NaN**
- **min(NaN, Y) = Y**
- **min(min(X , NaN), Y) = min(NaN, Y) = Y**
- **min(X,min(NaN,Y)) = min(X,Y)**

min(min(10, NaN), 100) = 100