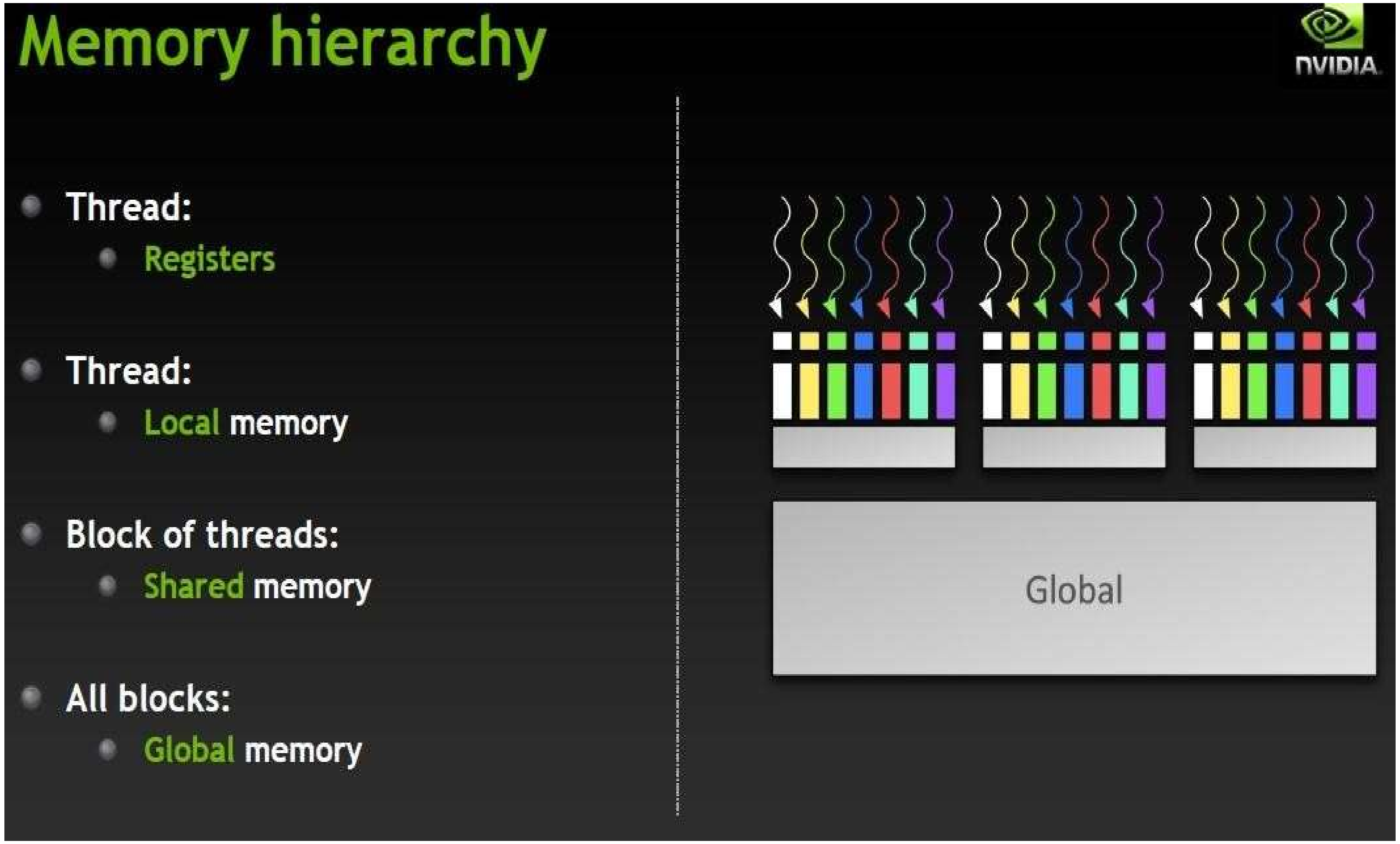
# MEMORY HANDLING

# NVIDIA GPU Memory Structures



## Memory hierarchy

- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory

NVIDIA
Global

## NVIDIA GPUs: Terminology

- Memory hardware
- Global Memory
  - DRAM available to all threads (SIMD processors in GPU)
- Local Memory
  - Private to the thread
- Shared Memory
  - Accessible to all threads of a Streaming Processor
- Thread Processor Registers

# CACHES

- A cache is a high-speed memory that is physically close to the processor core.
- Caches are expensive.
- The maximum speed of a cache is proportional to the size of the cache.
- L1 cache is the fastest, but is limited in size to usually around 16 K, 32 K, or 64 K. It is usually allocated to a single CPU core.
- The L2 cache is slower, but much larger, typically 256 K to 512 K.
- The L3 cache may or may not be present and is often several megabytes in size.
- The L2 and/or L3 cache may be shared between processor cores or maintained as separate caches linked directly to given processor cores.
- Generally, at least the L3 cache is a shared cache between processor cores on a conventional CPU.
- This allows for fast intercore communication via this shared memory within the device

## CACHES

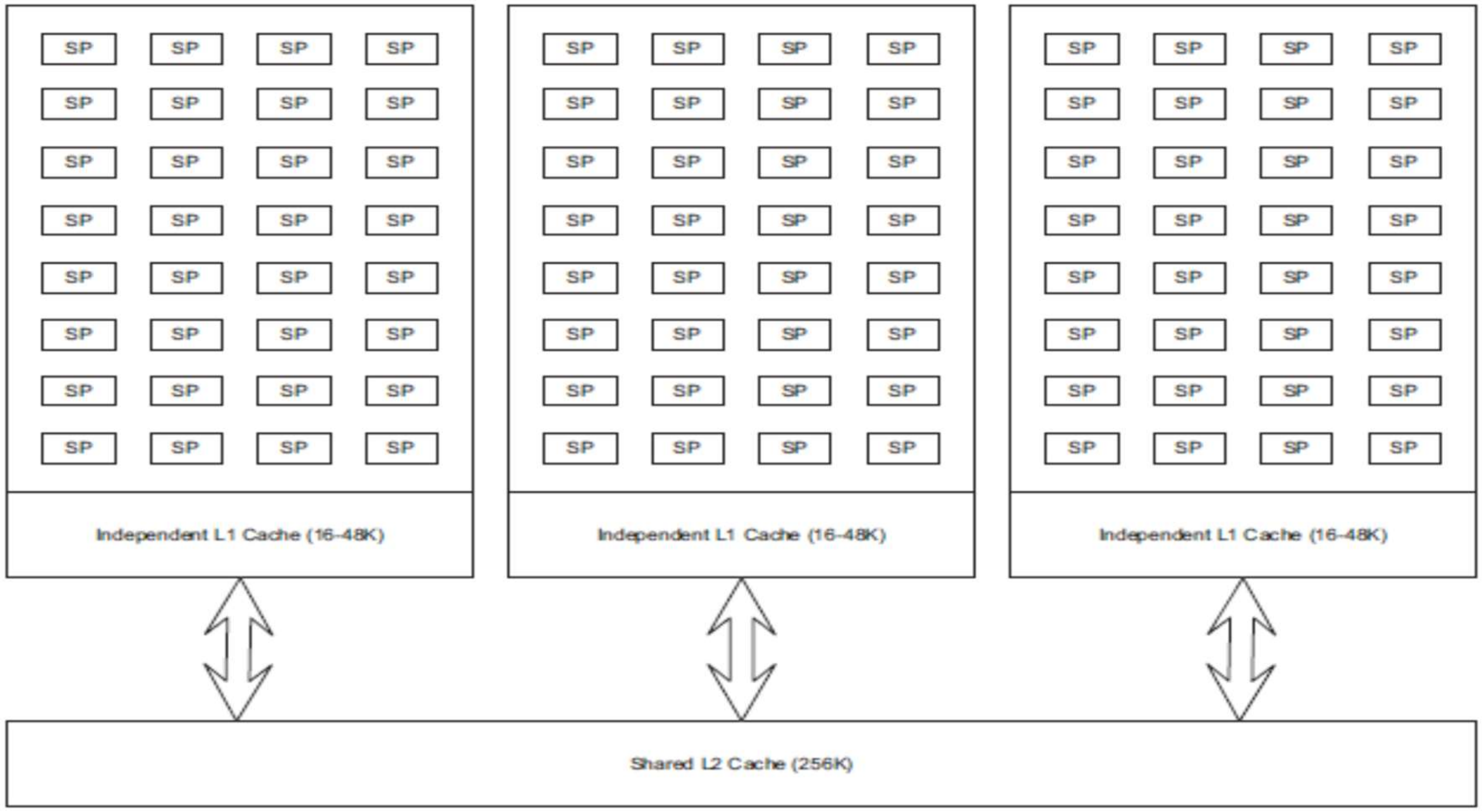


Fig: SM L1/L2 data path.

## CACHES

- In Fermi GPU  we introduce the concept of a nonprogrammer managed data cache.
- The architecture additionally has, per SM, an L1 cache that is both programmer managed and hardware managed.
- It also has a shared L2 cache across all SMs

## CACHES

- **Types of data storage**
- On a GPU, we have a number of levels of areas where you can place data, each defined by its potential bandwidth and latency, as shown in Table below.

# CACHES

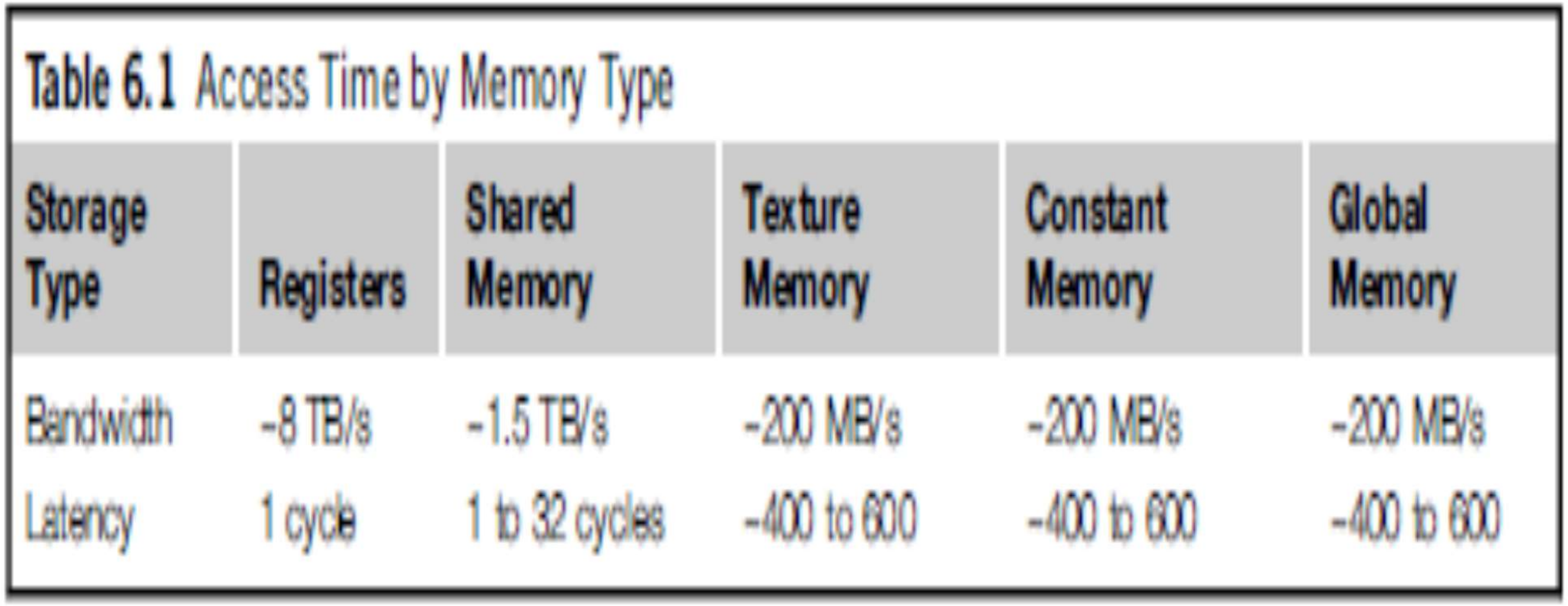


Table 6.1 Access Time by Memory Type

| Storage Type | Registers | Shared Memory | Texture Memory | Constant Memory | Global Memory |
|---|---|---|---|---|---|
| Bandwidth | -8 TB/s | -1.5 TB/s | -200 MB/s | -200 MB/s | -200 MB/s |
| Latency | 1 cycle | 1 to 32 cycles | -400 to 600 | -400 to 600 | -400 to 600 |

## REGISTER USAGE

- The GPU,  has thousands of registers per SM (streaming multiprocessor).
- An SM can be thought of like a multithreaded CPU core.
- On a typical CPU we have two, four, six, or eight cores.
- On a GPU we have N SM cores.
- On a Fermi GF100 series, there are 16 SMs on the top-end device. The GT200 series has up to 32 SMs per device
- A typical CPU will support one or two hardware threads per core.
- A GPU by contrast has between 8 and 192 SPs per core, meaning each SM can at any time be executing this number of concurrent hardware threads.

## REGISTER USAGE

- On GPUs, application threads are pipelined, context switched, and dispatched to multiple SMs.
- The number of active threads across all SMs in a GPU device is usually in the tens of thousands range.

## REGISTER USAGE

- One major difference we see between CPU and GPU architectures is how CPUs and GPUs map registers:
  - The CPU runs lots of threads by using register renaming and the stack.
  - To run a new task the CPU needs to do a context switch, which involves storing the state of all registers onto the stack (the system memory) and then restoring the state from the last run of the new thread.
  - This can take several hundred CPU cycles.
  - If you load too many threads onto a CPU it will spend all of the time simply swapping out and in registers as it context switches.
  - The effective throughput of useful work rapidly drops off as soon as you load too many threads onto a CPU.

## REGISTER USAGE

- The GPU by contrast is the exact opposite.
  - It uses threads to hide memory fetch and instruction execution latency
  - The GPU does not use register renaming, but instead dedicates real registers to each and every thread.
  - Thus, when a context switch is required, it has near zero overhead.
  - On a context switch the selector (or pointer) to the current register set is updated to point to the register set of the next warp that will execute.

## REGISTER USAGE

- We use registers to avoid usage of the slower memory types.
- For example, suppose we had a loop that set each bit in turn, depending on the value of some Boolean variable.

```
for (i=0; i<31; i++)
{
packed_result | = (pack_array[i] << i);
}
```

- Here we are reading array element ***i*** from an array of elements to pack into an integer, packed_ result.
- We're left shifting the Boolean by the necessary number of bits and then using a ***bitwise or*** operation with the previous result.

## REGISTER USAGE

- If the parameter ***packed _result*** exists in memory, you have **32 memory read and writes.**
- We can place the parameter ***packed_result*** in a local variable
- The compiler would place into a register.
- As we accumulate into the **register** instead of in main memory, and later write only the result to main memory, **we save 31 of the 32 memory reads and writes.**

## REGISTER USAGE

- Assume 500 cycles for one global memory read or write operation.
- For every value you'd need to read, apply the or operation, and write the result back.
- You have **32 X read + 32 X write = 64X500 cycles =32,000 cycles**.
- The register version would eliminate 31 read and 32 write operations, replacing the 500-cycle operations with single-cycle operations.

(1X memory read) +(1 X memory write)+(31x register read) + (31X register write)

or

**(1x500)+(1X500) +(31 X1)+ (31X 1)= 1062 cycles versus 32, 000 cycles.**

- This is a huge reduction in the number of cycles.
- We have a 31 times improvement to perform a relatively common operation in certain problem domains

## SHARED MEMORY

- Shared memory is effectively a user-controlled L1 cache.
- The L1 cache and shared memory share a 64 K memory segment per SM.
- In Kepler this can be configured in 16 K blocks in favour of the L1 or shared Memory.
- In Fermi the choice is 16K or 48K in favour of the L1 or shared memory.
- The shared memory has in the order of 1.5 TB/s bandwidth with extremely low latency.
- This is hugely superior to the up to 190GB/s available from global memory, but around one-fifth of the speed of registers.

## SHARED MEMORY

- Shared memory is a bank-switched architecture.
- On Fermi it is 32 banks wide, and on G200 and G80 hardware it is 16 banks wide.
- Each bank of data is 4 bytes in size, enough for a single-precision floating-point data item or a standard 32-bit integer value.
- Kepler also introduces a special 64 bit wide mode .
- There is no need for a one-to-one sequential access, just that every thread accesses a separate bank in the shared memory.
- There is a crossbar switch connecting any single bank to any single thread.
- This is very useful when you need to swap the words.

## SHARED MEMORY

- Every thread in a warp reads the same bank address.
- This triggers a broadcast mechanism to all threads within the warp.
- Usually thread zero writes the value to communicate a common value with the other threads in the warp (fig below).
- If we have any other pattern, we end up with bank conflicts of varying degrees.
- This means you stall the other threads in the warp that idle while the threads accessing the shared memory address queue up one after another.
- One important aspect of this is that it is not hidden by a switch to another warp, so we do in fact stall the SM.