Applied Parallel Computing
parallel-computing.pro

# Memory Hierarchy

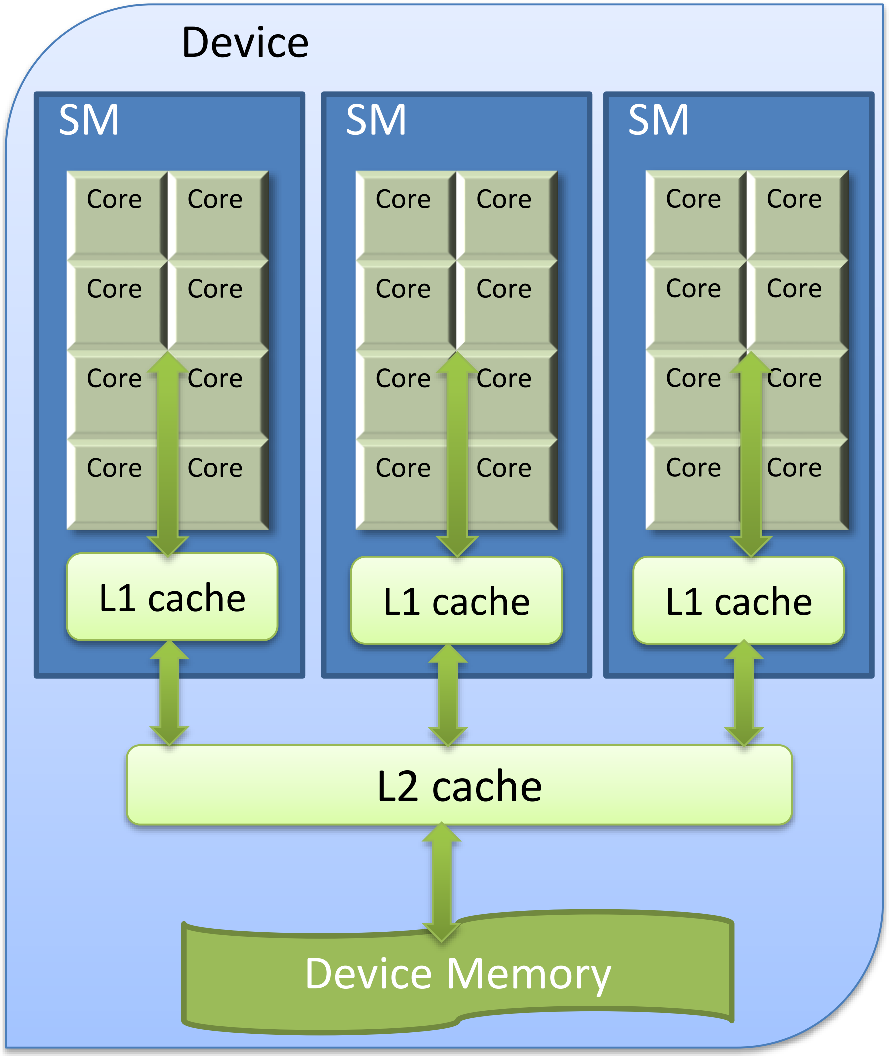
Pavel Yakimov

Applied Parallel Computing
parallel-computing.pro

## Global Memory

# Global Memory

- Located in **DRAM GPU**
- Up to 16Gb
  - Can be checked by totalGlobalMem
- **Cacheable,** uses caches L1 and L2:
  - L1 – located in each SM
    maximum size - 64KB
    minimal size - 0KB
  - L2 – on device
    maximum size 6 MB
    Device parameter l2CacheSize



## L1 Management

- Caching to L1 can be switched off **at compilation**
  - nvcc -Xptxas -dlcm=ca

    enabled L1 caching (by default)
  - nvcc -Xptxas -dlcm=cg

    disabled L1 caching, in binary code all global memory accesses are translated to instructions that don’t use L1 while executing !
- The differences are in binary code, not in execution mode

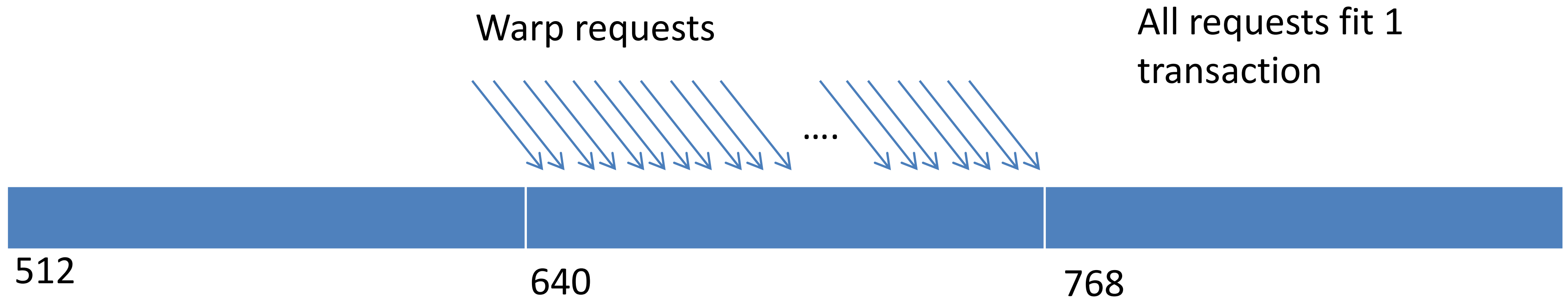
## Transactions

- Access to global memory:
  - With enabled cache L1 – transactions of 128 bytes
    - ✓ Cache line L1 – 128 bytes
  - With disabled cache L1 – transactions of 32 bytes
    - ✓ Cache line L2 – 32 bytes

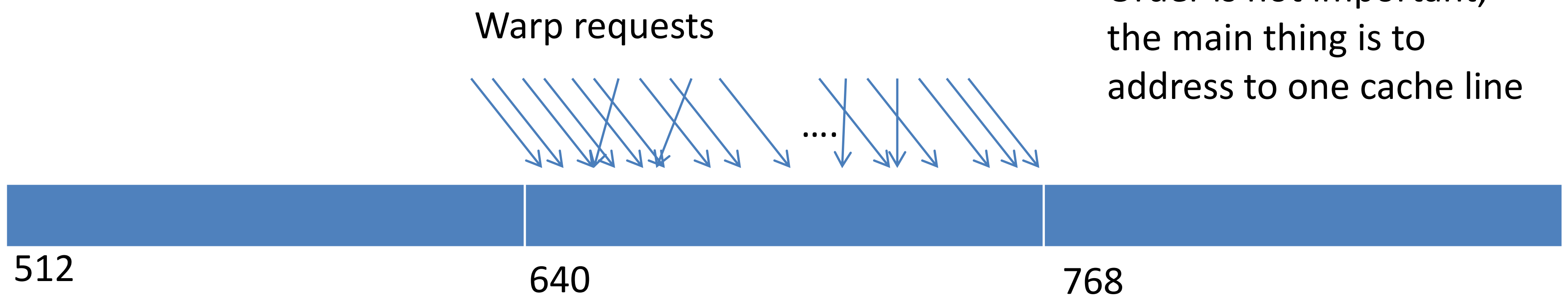
- Transactions are aligned by size(naturally aligned)

- Memory access instruction is performed simultaneously for all threads in a warp (**SIMT**)
  - Threads can access to different elements
  - Performing as many transactions, as needed to cover memory requests of all threads in a warp

# Access Patterns: L1 enabled

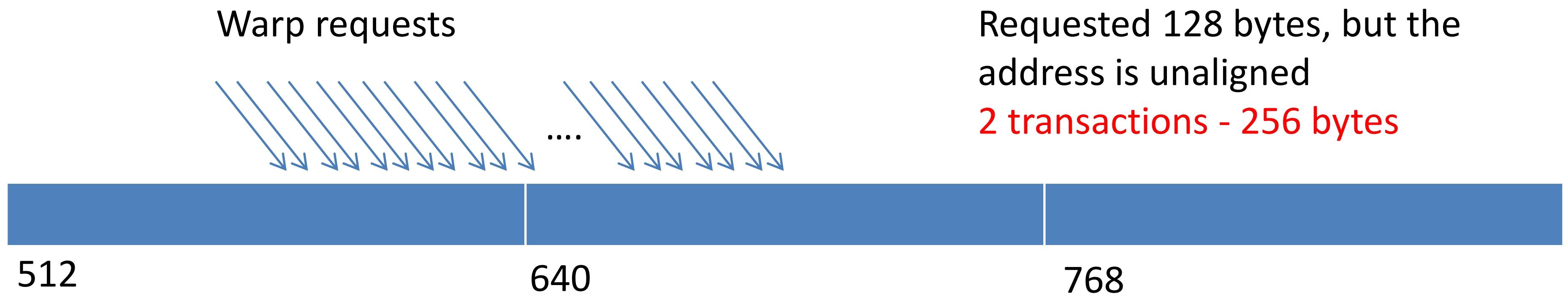


All requests fit 1 transaction

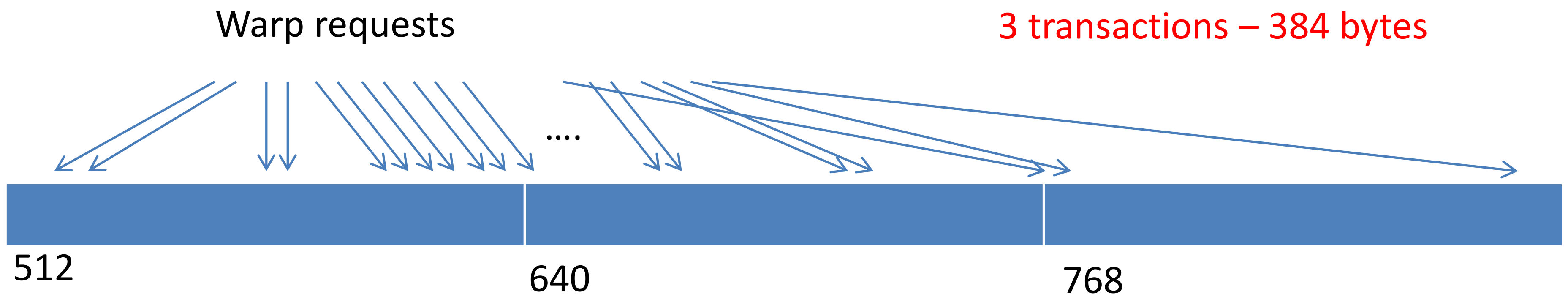


Order is not important, the main thing is to address to one cache line

# Access Patterns: L1 enabled

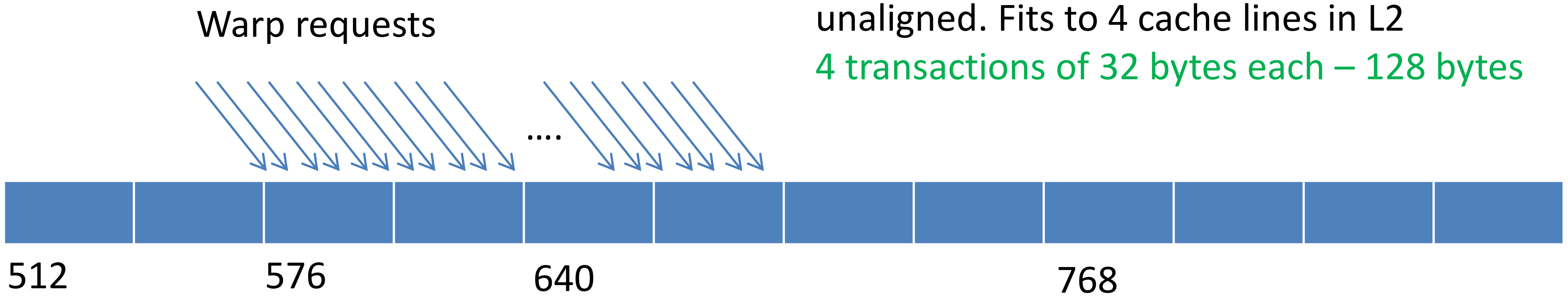


Requested 128 bytes, but the address is unaligned
2 transactions - 256 bytes



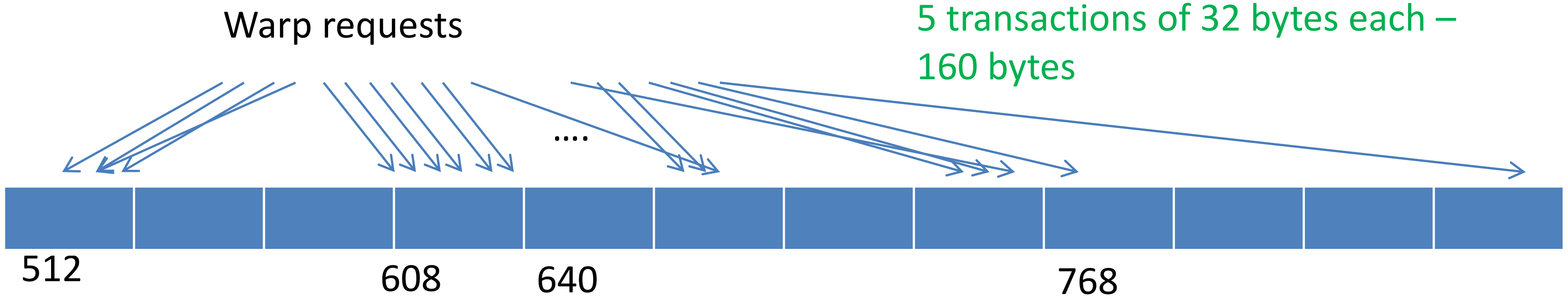
Requested 128 bytes, but requests are scattered within 3 cache lines -
3 transactions – 384 bytes

# Access Patterns: L1 disabled



Requests 128 bytes, but the address is unaligned. Fits to 4 cache lines in L2

4 transactions of 32 bytes each – 128 bytes



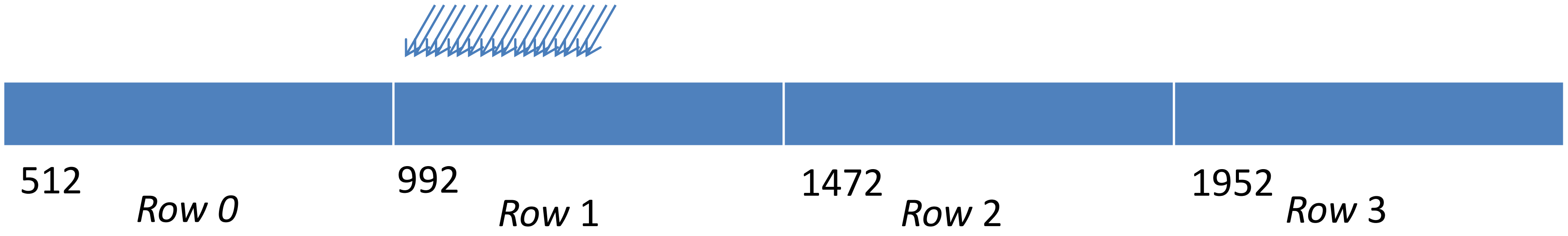
Requests 128 bytes, but requests are scattered within 5 cache lines

5 transactions of 32 bytes each – 160 bytes

# Matrices in Global Memory

- A matrix is stored linearly, row-by-row
- Let the length of a row – 480 bytes (120 float)
  - access – matrix[idy*120 + idx]

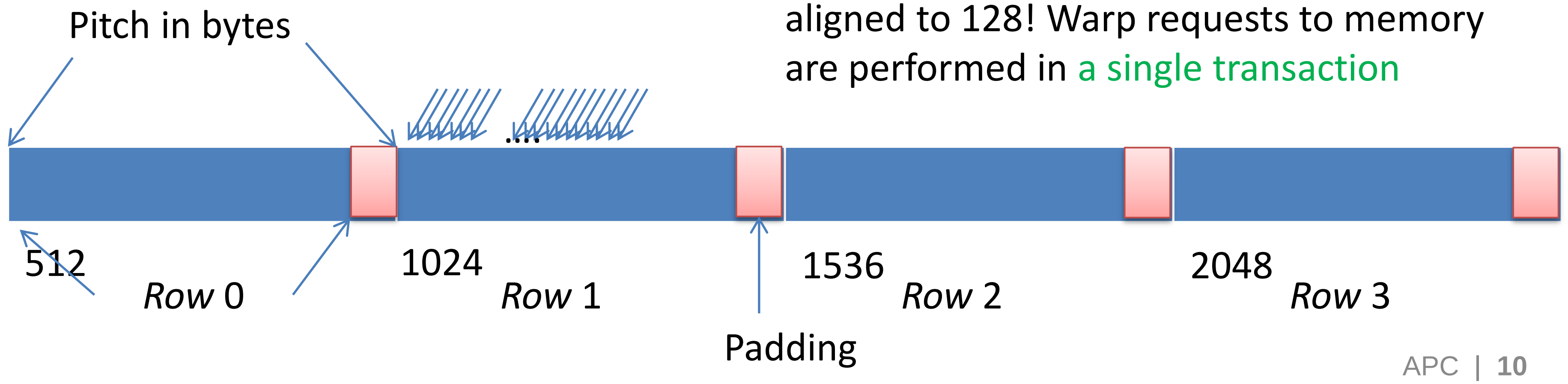
Address of each row beginning, except the first one, is not aligned to 128 bytes–
2 transactions



## Matrices in Global Memory

- Supplement each row to a multiple of 128 bytes – in this example, 480 + 32 = 512,
  32 bytes is pitch – actual width in bytes
- These bytes can not be used, i.e. 32/512=6% extra memory will be allocated (but for large matrices, this share will be significantly less)
- However, each row will be aligned to 128 bytes
  - Request matrix[idy*128+ idx]

Address of the beginning of each line is aligned to 128! Warp requests to memory are performed in a single transaction



## Memory Allocation with Padding

- `cudaError_t cudaMallocPitch (void ** devPtr, size_t * pitch, size_t width, size_t height)`
  - width – logical matrix width in bytes
  - Allocates not less than width * height bytes , can add some padding to the end of a row, in order to align the beginning of rows
  - Stores the pointer to memory (*devPtr)
  - Stores actual width of rows to (*pitch)

- Matrix element address (Row, Column), allocated with cudaMallocPitch:

```
T* pElement = (T*)((char*) devPtr + Row * pitch) + Column;
```

!

## Copying to Matrix with Padding

- `cudaError_t cudaMemcpy2D ( void* dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind )`
  - dst - a pointer to the matrix you want to copy ,

    dpitch – ***actual*** row width in bytes
  - src - a pointer to the matrix from which you want to copy,

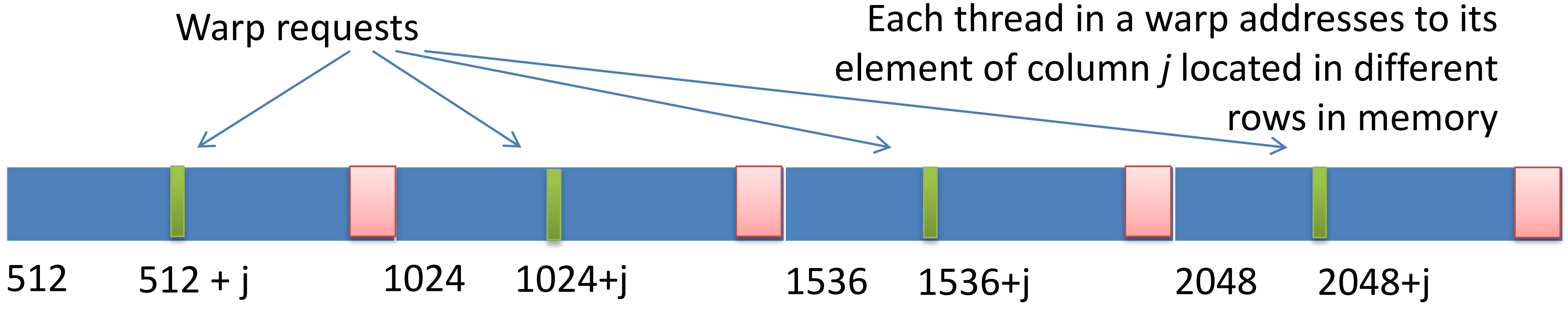
    spitch – ***actual*** row width in bytes
  - width – width of matrix transfer (columns in bytes)
  - height – Height of matrix transfer (rows)
  - kind – type of transfer (similar to cudaMemcpy)
- This function width bytes from the beginning of each row of source matrix. Total transfer size is width*height bytes, herewith
  - Address of row with index **Row** is calculated with actual width:

    `(char*)src + Row* spitch` – *source matrix*

    `(char*)dst + Row* dpitch` – *destination matrix*

## How to Reference to Matrix by Columns?

- Matrix is allocated by rows, and the memory requests are by columns



Each thread in a warp addresses to its element of column *j* located in different rows in memory

If the matrix has a size greater than 128 bytes, then these requests would never fit to a single transaction!

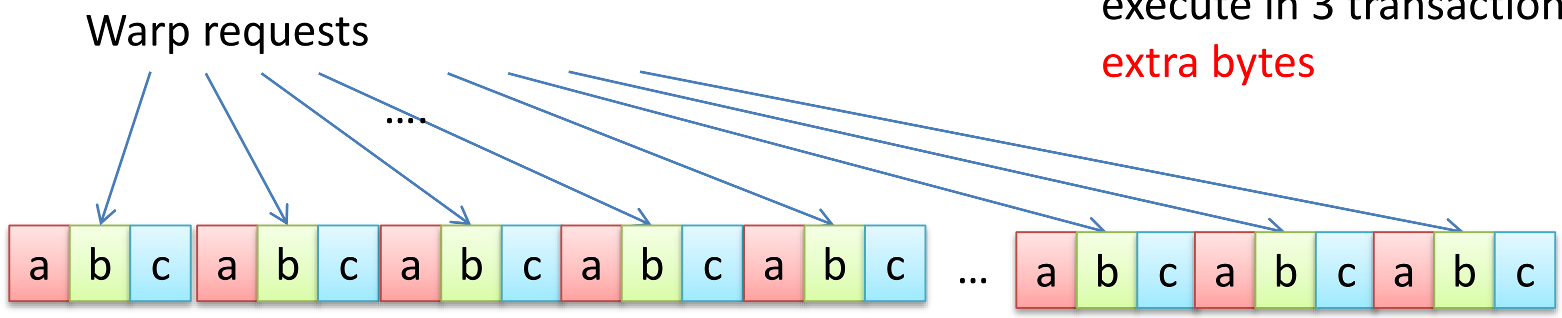
## Transpose!

- Solution – store transposed matrix!
    - This leads into actual sequential addressing to rows in memory just like they are columns
    - The memory for transposed matrix should also be allocated with cudaMallocPitch

## Array of Structs?

```
struct example {
   int a;
   int b;
   int c;
}
__global__ void kernel(example * arrayOfExamples) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    arrayOfExamples[idx].c =
         arrayOfExamples[idx].b + arrayOfExamples[idx].a;
}
```

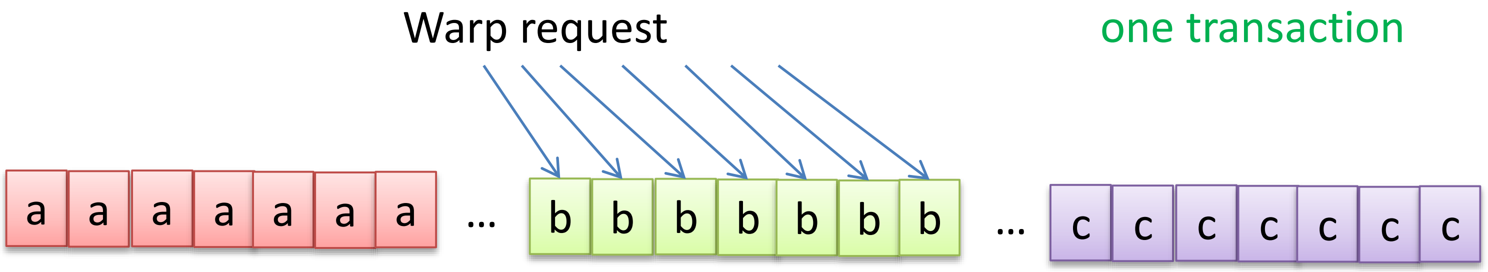
Warp addressing to memory will execute in 3 transactions - 256 extra bytes



# Struct of Arrays!

```
struct example {
   int *a;
   int *b;
   int *c;
}
__global__ void kernel(example arrayOfExamples) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    arrayOfExamples.c[idx] =
         arrayOfExamples.b[idx] + arrayOfExamples.a[idx];
}
```

Warp memory requests fit to one transaction



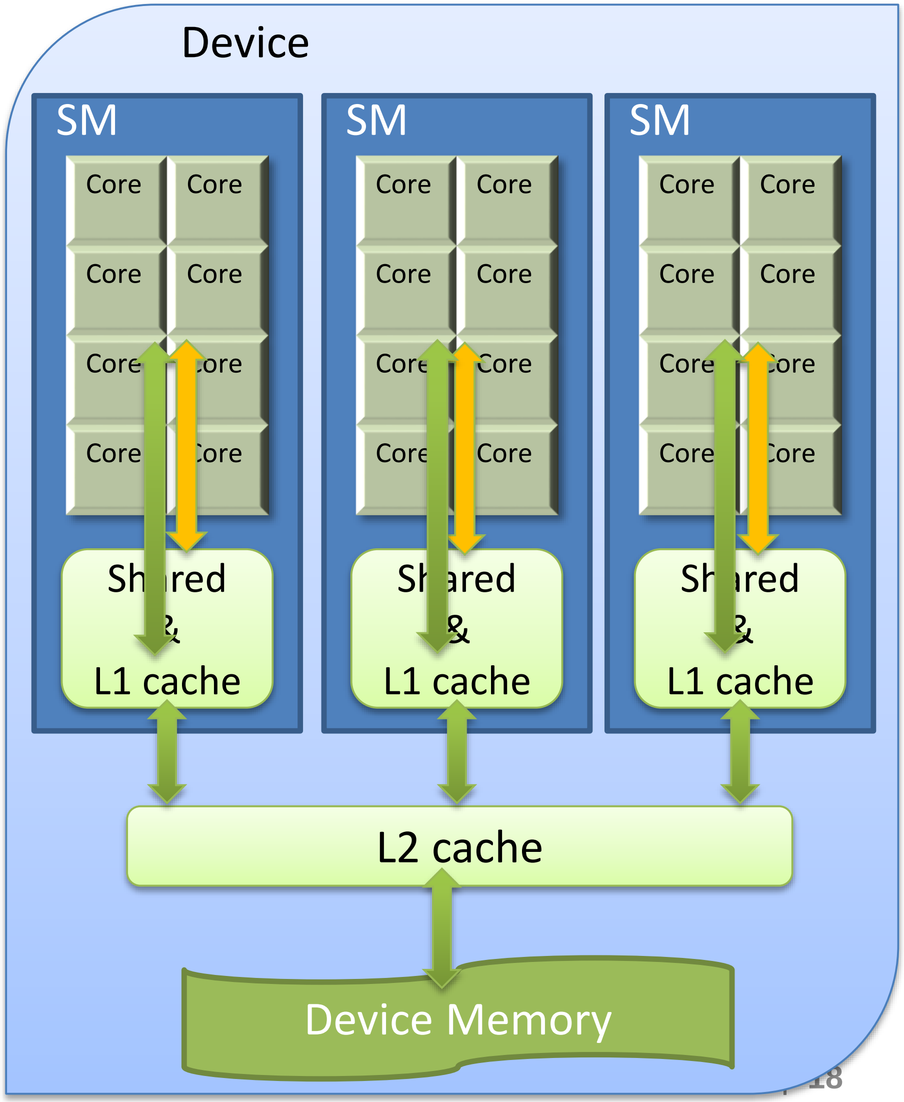
Applied Parallel Computing
parallel-computing.pro

## Shared Memory

# Shared Memory

- Located in the same device in SM as L1
- Jointly used by all threads in a block
- If a multiprocessor executes several blocks - memory is divided equally between them
- Each block gets its limited address space of shared memory

- Configurations:
  - 16KB shared memory, 48KB L1
  - 32KB shared memory, 32KB L1
  - 48KB shared memory, 16KB L1 – by default



# Shared Memory Allocation

- Statically:
  - In GPU code declare a static array or a variable with qualifier `__shared__`

```
#define SIZE 1024
__shared__ int array[SIZE]; //array
__shared__ float varSharedMem; //variable
```

- Dynamically:
  - In GPU code declare a pointer to array in shared memory :

```
extern __shared__ int array[];
```

  - In host code specify how much shared memory in bytes should be additionally allocated per each block. The third parameter of kernel launch passes this value

```
kernel<<<gridDim, blockDim, SIZE >>>(params)
```

## Features of Use

- In terms of programming, variables with qualifier __shared__ :

  - Can be declared in the global scope or within functions
    - When declared in a function perform as static, i.e. one instance exists for all function calls

  - Individual for each block and attached to their personal space of shared memory
    - each block of threads sees ‘his’ value
  - Exist only for the lifetime block
    - not available from host or from other blocks
  - Can not be initialized when declaring

## Features of Use

- Shared memory is statically allocated in kernel only if it uses any `__shared__` variables, declared without [] in
  - the global scope
  - in kernel
  - In functions, called from kernel,
- Kernel can be simultaneously provided by both dynamically and statically allocated shared memory
- All variables `extern __shared__ type var[]` point to the beginning dynamic shared memory allocated to block

## Synchronization

- Consider the example of kernel running on a linear one-dimensional grid :

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Each thread
  - Calculates __sinf from its index and stores it to the corresponding element of the output array
  - Reads an element written by neighbor thread

## Synchronization

- Consider the example of kernel running on a linear one-dimensional grid:

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Warp execute in unpredictable order
  - It may happen that a thread has not yet written a value, and the neighboring thread already tries to read it!
  - *read-after-write, write-after-read, write-after-write* conflicts

## Synchronization

- For explicit synchronization some built-in functions are provided :

  - `void __syncthreads();`

    When you call this function, thread waits until:

    ✓ all threads in a block reach this point

    ✓ the results of all currently initiated operations with global / shared memory will be visible to all threads of the block

- __syncthreads() can be invoked in the branches of ‘if’ statement only if the result of its condition is the same for all threads in a block,
  - if not then execution may deadlock or become unpredictable

## Synchronization

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    __syncthreads();
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Each thread
  - Calculates __sinf from its index and stores it to the corresponding element of the output array
  - Wait until all threads in a block complete calculations
  - Reads an element written by neighbor thread

## Strategy of Usage

- Shared memory can be considered as a cache, **controlled by a programmer**
  - Low latency – located on the same chip as L1, speed of memory requests is comparable to registers
  - Application explicitly allocates and uses shared memory
  - Access pattern can be arbitrary, unlike in L1
- Even if the hardware cache L1 is able to process all requests (L1 load hit ~ 100%), the use of shared memory allows making full use of the equipment
  - Otherwise 16KB (or 48KB, if you forgot to set the proper mode) fast shared memory is idle

## Strategy of Usage

- Typical strategy:
  - Threads in a block collectively:
    1. Download data from global memory to shared
       - Each thread executes part of this downloading
    2. Synchronize
       - That no thread starts reading the data being uploaded by another thread, before it finishes uploading
    3. Use the downloaded data to calculate results
       - If threads write something to shared memory, it may also need to synchronize again
    4. Write the results to global memory

# Example of Shared Memory Usage

```
__global__ void kernel(int sizeOfArray1, int sizeOfArray2, int *devPtr, int *res)
{
    extern __shared__ int dynamicMem[];  // a pointer to dynamic shared memory
    __shared__ int staticMem[1024];      // static array in shared memory
    __shared__ int var;                 // a variable in shared memory
    int *array1 = dynamicMem; // address of the first array in dynamic shared memory
    int *array2 =
         array1 + sizeOfArray1;  // address of the second array in dynamic shared memory

    staticMem[threadIdx.x] = devPtr[threadIdx.y]; // loading data to shared memory
   __syncthreads();  // wait until all threads finish loading

    array2[threadIdx.x] =
         2 * staticMem[(threadIdx.x - 10) % blockDim.x];  // access to an element
                                                          //written by another thread
    __syncthreads();  // wait until all threads finish writing
    res[threadIdx.x] = array2[threadIdx.x];  // write the results to global memory
}
```

# Example of Shared Memory Usage

```
__global__ void kernel(int sizeOfArray1, int sizeOfArray2, int *devPtr, int *res)
{
    extern __shared__ int dynamicMem[]; // a pointer to dynamic shared memory
    __shared__ int staticMem[1024];     // static array in shared memory
    __shared__ int var;                 // a variable in shared memory
    int *array1 = dynamicMem; // address of the first array in dynamic shared memory
    int *array2 =
         array1 + sizeOfArray1;  // address of the second array in dynamic shared memory

    staticMem[threadIdx.x] = devPtr[threadIdx.y]; // loading data to shared memory
    __syncthreads(); // wait until all threads finish loading

    array2[threadIdx.x] =
        2 * staticMem[(threadIdx.x - 10) % blockDim.x]; // access to an element
                                                        // written by another thread
    // Actually, there is no need in synchronization here
    res[threadIdx.x] = array2[threadIdx.x]; // write the results to global memory
}
```

## Example of Kernel Launching with Dynamic Shared Memory

- Host code

```
int *devPtr;
cudaMalloc(&devPtr, 1024*sizeof(int));
kernel<<<3,1024,1024*sizeof(int)>>>(512,512, devPtr);  // launch three blocks of
                                      //1024 threads. Dynamically allocate 4KB of shared memory per block
```

Size of dynamically allocated memory, in bytes per block

- If the total (static + dynamic) requested shared memory size exceeds the total available memory (16KB, 32KB or 48KB), an error will occur

## Writing to Shared Memory

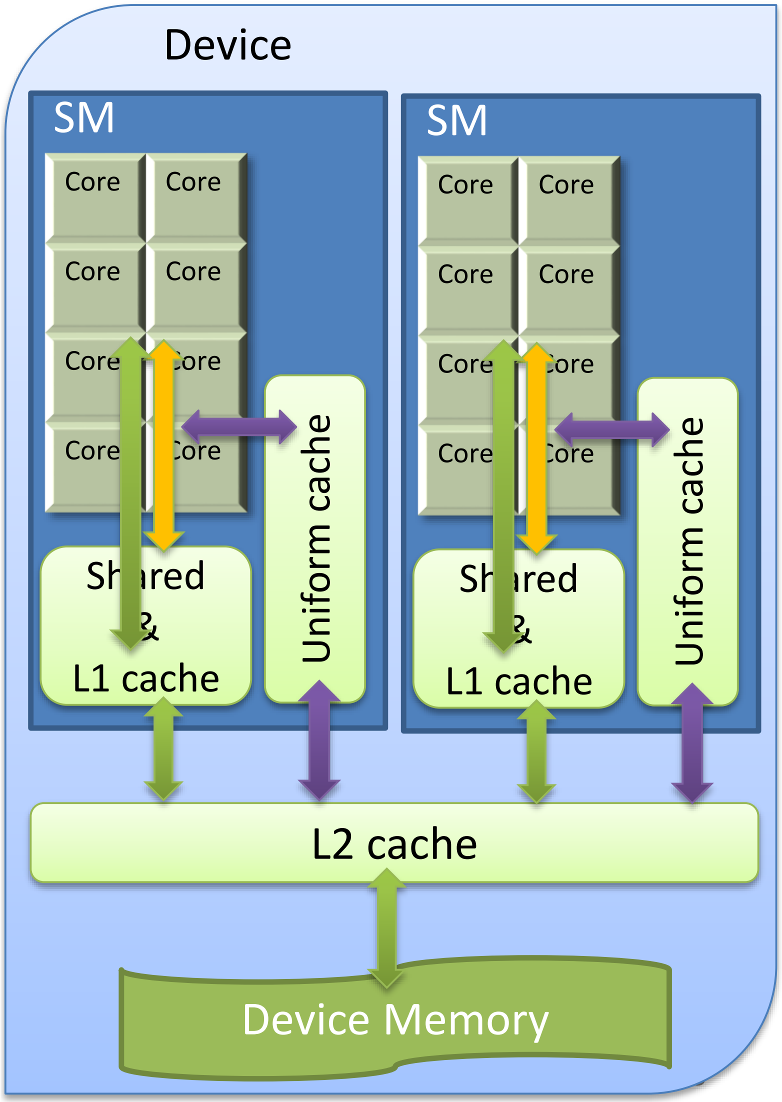
- Several threads of a warp trying to write to the same address
  - The operation will be executed only by one thread
  - Which – unknown
    - ✓ (Apparently - the latter thread of a warp, from those that have to write)
    - ✓ The result is unpredictable
      - At least, because the order is unpredictable and no one knows what warp will be the latest

Applied Parallel Computing
parallel-computing.pro

## Constant Memory

# Constant Memory

- Located in **DRAM GPU**
- Size up to 64KB
  - Device parameter **totalConstMem**
- **Cacheable** in special read-only cache – Unifrom Cache
  - Up to 8 KB



## Declaration

- In the global scope

```
__constant__ int constMem[1024];
__constant__ int constVar;
```

- You can additionally specify `__device__` , to indicate that the memory is allocated on the device:

```
__device__ __constant__ int constVar2;
```

## Features

- Allocated at application startup, released at the end of application lifetime
- Is available for reading (read-only!) from any thread of any grid in a usual way:

```
__constant__ int constMem[32];
__global__ void kernel() {
    …
    int a = constMem[ threadIdx.x / 32 ];
    …
}
```

- Available from host with special functions from toolkit: `cudaGetSymbolAddress() /cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()`

## Example

```
__constant__ float constData[256];
```

- Host:

```
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```

# Requests to Constant Memory

- Request is performed simultaneously for all threads in a warp (SIMT)
- The initial requests is divided into as many queries as many different addresses it contains
  - Each query is performed either through a request to cache, in case of cache hit, or to global memory
  - If there were *n* queries, then the bandwidth is reduced by *n* times

```
__constant__ int constMem[32];
__global__ void kernel() {
   …
   int a = constMem[ threadIdx.x / 32 ]; // 1 request to constant
                                           memory per warp
   int a = constMem[ threadIdx.x]; // 32 request to constant
                                           memory per warp

   …
}
```

## Uniform Access

- In addition to processing requests to constant memory, Uniform Cache also processes the Uniform Accesses - when all threads of a single warp accesses global memory at the same address
  - It is possible only when:
    - ✓ The access is read-only
    - ✓ The address doesn't depend on the thread's index (**`threaIdx`**)

```
while(k < 100 ) tmp += a[blockIdx.x + k++];
```

In assembler, compiler will change an ordinary instruction of load from global memory to the instruction of uniform load, which will be executed using Uniform Cache

- The second requirement guarantees that all threads of a warp request to the same address
- To help the compiler with the first requirement, we can add the qualifier **`const`** to the pointers

## Passing Parameters and Grid to Kernels

- Parameters are passed to the kernel through the constant memory
  - Parameters are passed in a single copy for all threads of grid
  - This is acceptable, because,
    - ✓ basically, threads of a warp are requesting the same parameter -> Uniform Access
    - ✓ after the first warp, the parameters will already be in the cache
- The total size the passed parameters must be no larger than 4 KB

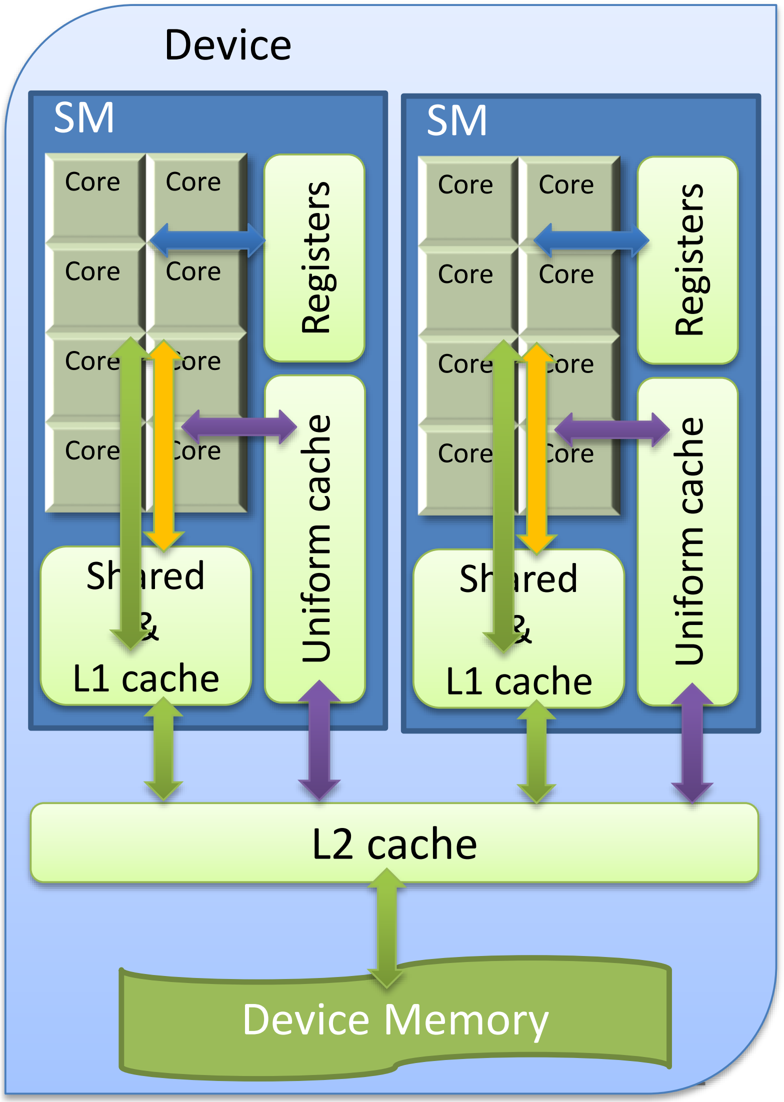
- Grid size (`gridDim, blockDim)` is also passed through the constant memory :
  - Thread receives `threadIdx` and `blockIdx` from special registers
  - `gridDim`, `blockDim` are **read from the constant memory** at the very beginning of execution (Uniform)

Applied Parallel Computing
parallel-computing.pro

## Registers and Local memory

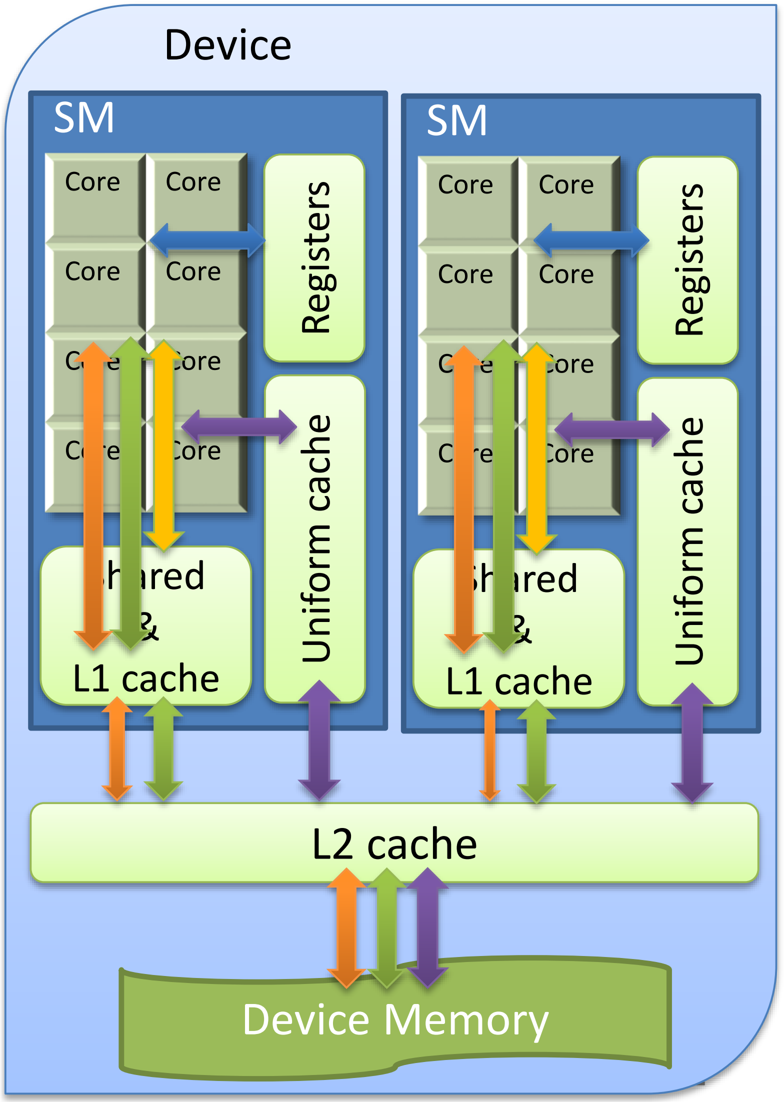
## Registers

- Located on SM
- The fastest memory access
- Each multiprocessor contains 64K 32-bit registers
    - 256 KB of registers
    - Device parameter - **regsPerBlock**
- One thread can have up to 63 registers
- Distributed **during compilation**
- Each thread is an exclusive user of its registers for the duration of the kernel execution. Access to the registers of other threads is restricted



## Local Memory

- Located in **DRAM**
- Access is made by the same rules as requests to the global memory
  - Caching to L1
  - Transactions
- Unavailable explicitly
- Has simplified addressing scheme
  - Optimized to minimize the number of transactions



## When is local memory used?

- Typically, the compiler places into registers all local variables
- But there are exceptions, which are placed into the local memory
  - Arrays, for which you can not always determine which element in what period of time is being accessed (not constant indexes)
  - Large arrays or structures that would use too many registers
  - Any variable, if the limit of 63 registers per thread is exceeded (register spilling)
- Some built-in math functions can use the local memory
- Local memory is used to pass a part of the operands for function call
  - Stack frame for recursive calls is modeled in the local memory

## nvcc -Xptxas -v

- Displays the number of registers, constant memory, local memory and static shared memory used by the kernel:

```
pyakimov@localhost:~/programming/testMod$ nvcc -arch=sm_20 -Xptxas -v test.cu
ptxas info    : 0 bytes gmem, 8 bytes cmem[2]
ptxas info    : Compiling entry function '_Z13matmul_kernelv' for 'sm_20'
ptxas info    : Function properties for _Z13matmul_kernelv
    8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 4 bytes smem, 32 bytes cmem[0]
```

## Thank You!

## L1 Cache Configuration

- Cache supports 3 configuration modes: 48KB, 32KB , and 16KB
- Switching modes:

  - `cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig)`

    cacheConfig resets cache mode on the whole device
    - ✓ Possible modes:
      - cudaFuncCachePreferNone – without preferences (by default). Sets the last used configuration . The initial configuration – 16KB L1
      - cudaFuncCachePreferShared: 16KB L1
      - cudaFuncCachePreferEqual: 32KB L1
      - cudaFuncCachePreferL1: 48KB L1

  -
    ```
    cudaError_t cudaFuncSetCacheConfig ( const void* func,
                                         cudaFuncCache  cacheConfig)
    ```

    Switches cache to cacheConfig configuration for function func
    - ✓ By default - cudaFuncCachePreferNone

## L1 Cache Configuration

- It is just ‘preferences’
  - Runtime may not provide the preset configuration mode depending on kernel and device parameters
- If a kernel doesn’t use shared memory, then it is favorable to switch to cudaFuncCachePreferL1 mode

- In general, it is worth checking the performance of all modes: !
  - (-dlcm=**ca**, -clcm=**cg**)x(**16**KB, **32**KB, **48**KB)