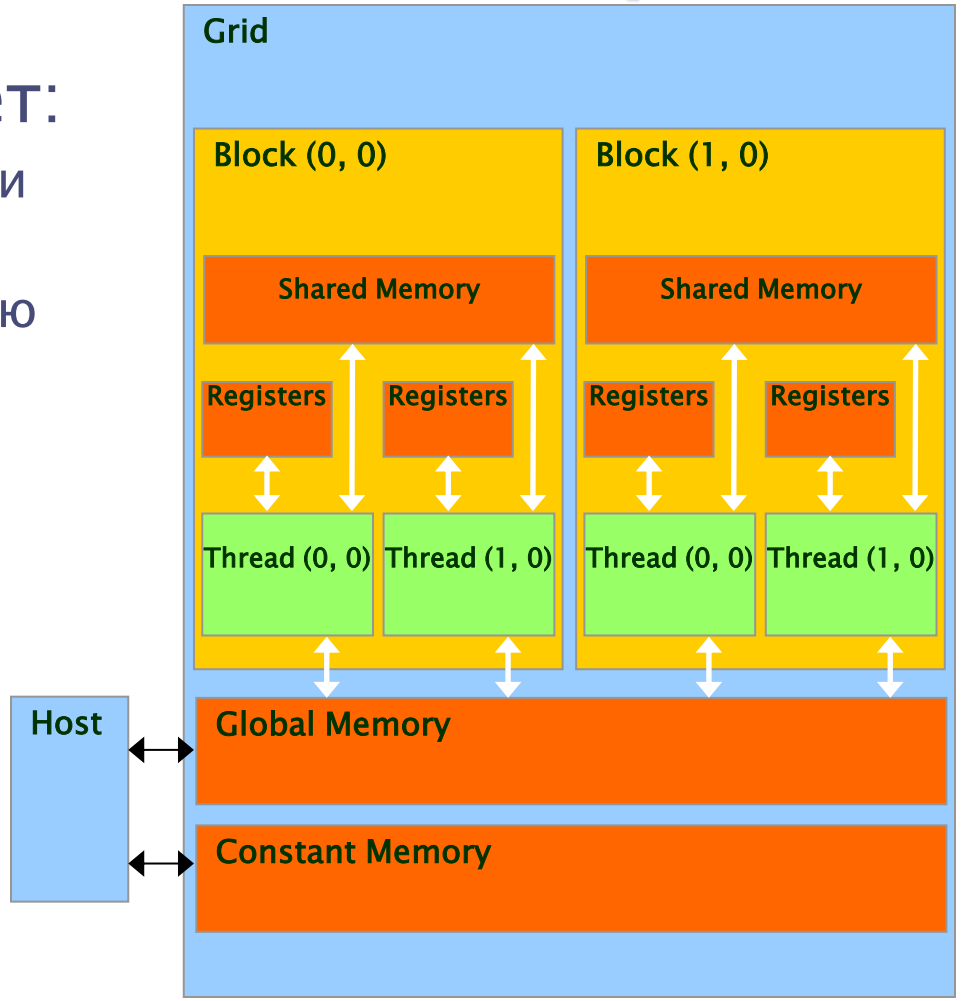
Многоуровневая память видеокарты

Оптимизация доступа к памяти

Реализация хранения переменных на видеокарте

- Каждый поток может:
  - Читать/записывать свои регистры
  - Читать/записывать свою локальную память
  - Читать/записывать разделяемую память блока
  - Читать/записывать общую глобальную память
  - Читать общую константную память



Квалификаторы переменных

Объявление переменной	Память	Видимость	Время жизни
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- “автоматические” скалярные переменные без квалификатора помещаются на регистры
  - в случае недостатка регистров – выгружаются в локальную память потока
- “автоматические” массивы без квалификатора размещаются в локальной памяти потока

Время доступа к переменным

Объявление переменной	Память	Замедление
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- скалярные переменные размещаются на регистрах
- разделяемые переменные – в быстрой памяти на чипе
- локальные массивы и глобальные переменные – в глобальной видео памяти
- константы – в кешируемой константной памяти

Масштабируемость

Объявление переменной	Кол-во экземпляров	Видимость
`int var;`	100,000	1
`int array_var[10];`	100,000	1
`__shared__ int shared_var;`	100	1000
`__device__ int global_var;`	1	100,000
`__constant__ int constant_var;`	1	100,000

Выделение локальной памяти: переменные внутри функции

```
// motivate per-thread variables with
// Ten Nearest Neighbors application
__global__ void ten_nn(float2 *result, float2 *ps, float2 *qs,
                       size_t num_qs)
{
  // p goes in a register
  float2 p = ps[threadIdx.x];

  // per-thread heap goes in off-chip memory
  float2 heap[10];

  // read through num_qs points, maintaining
  // the nearest 10 qs to p in the heap
  ...
  // write out the contents of heap to result
  ...
}
```

Выделение разделяемой памяти: shared variables

```
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads
  // complete before continuing
  __syncthreads();
  ...
}
```

Динамическое выделение разделяемой памяти

```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
  extern __shared__ int s_data[];
  ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block_size * sizeof(int)>>>(r,i);
```

А если массивов несколько?

```
extern __shared__ char data[];

__global__ void adj_diff(int *result, int n1, int n2, int n3)
{
 short* array1 = (short*)data;
 int* array2 = (int*)data[n1*2]; //ошибка выравнивания, если n1 - нечетное
 float* array3 = (float*) data[n1*2+n2*4];
  ...

}
```

Выделение глобальной памяти

- Динамически с хоста через `cudaMalloc()`
- Статически – глобальная переменная с атрибутом `__device__`
- Динамически из ядер через `malloc()`

Динамически с хоста

```
__global__ void kernel(int *arrayOnDevice) {
     arrayOnDevice[threaIdx.x] = threaIdx.x;
}

int main() {
     size_t size = 0;
     void *devicePtr = NULL;
     int hostMem[512];
     cudaMalloc(&devicePtr, sizeof(hostMem));
     cudaMemcpy(devicePtr, hostMem, size, cudaMemcpyHostToDevice);
     kernel<<<1,512>>>(devicePtr);
}
```

Статически

```
__device__ int arrayOnDevice[512]
__global__ void kernel() {
     arrayOnDevice[threaIdx.x] = threaIdx.x;
}

int main() {
    size_t size = 0;
    void *devicePtr = NULL;
    int hostMem[512];
    cudaGetSymbolSize(&size, arrayOnDevice);
    cudaMemcpyToSymbol(arrayOnDevice, localMem, size);
    kernel<<<1,512>>>(devicePtr);
}
```

Статически

```
__device__ int arrayOnDevice[512]
__global__ void kernel() {
     arrayOnDevice[threaIdx.x] = threaIdx.x;
}

int main() {
     size_t size = 0;
     int hostMem[512];
     void *devicePtr = NULL;
     cudaGetSymbolSize(&size, arrayOnDevice);
     cudaGetSymbolAddress(&devicePtr, arrayOnDevice);
     cudaMemcpy(devicePtr, hostMem, size, cudaMemcpyHostToDevice);
     kernel<<<1,512>>>();
}
```

cuda*Symbol*

- Переменные с атрибутами `__device__` и `__constant__` находятся в глобальной области видимости и хранятся в объектном модуле как отдельные символы

- Память под них выделяется статически при старте приложения, как и под обычные глобальные переменные

- Работать с ними на хосте можно через функции `cudaMemcpyToSymbol , cudaMemcpyToSymbolAsync, cudaGetSymbolAddress, cudaMemcpyFromSymbol, cudaMemcpyFromSymbolAsync, cudaGetSymbolSize`

Динамически из ядер

```
#include <stdlib.h>

__global__ void kernel() {
    size_t size = 1024 * sizeof(int);
    int *ptr = (int *)malloc(size);
    memset(ptr, 0, size)
    free(ptr)
}

int main() {
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    kernel<<<1, 128>>>();
}
```

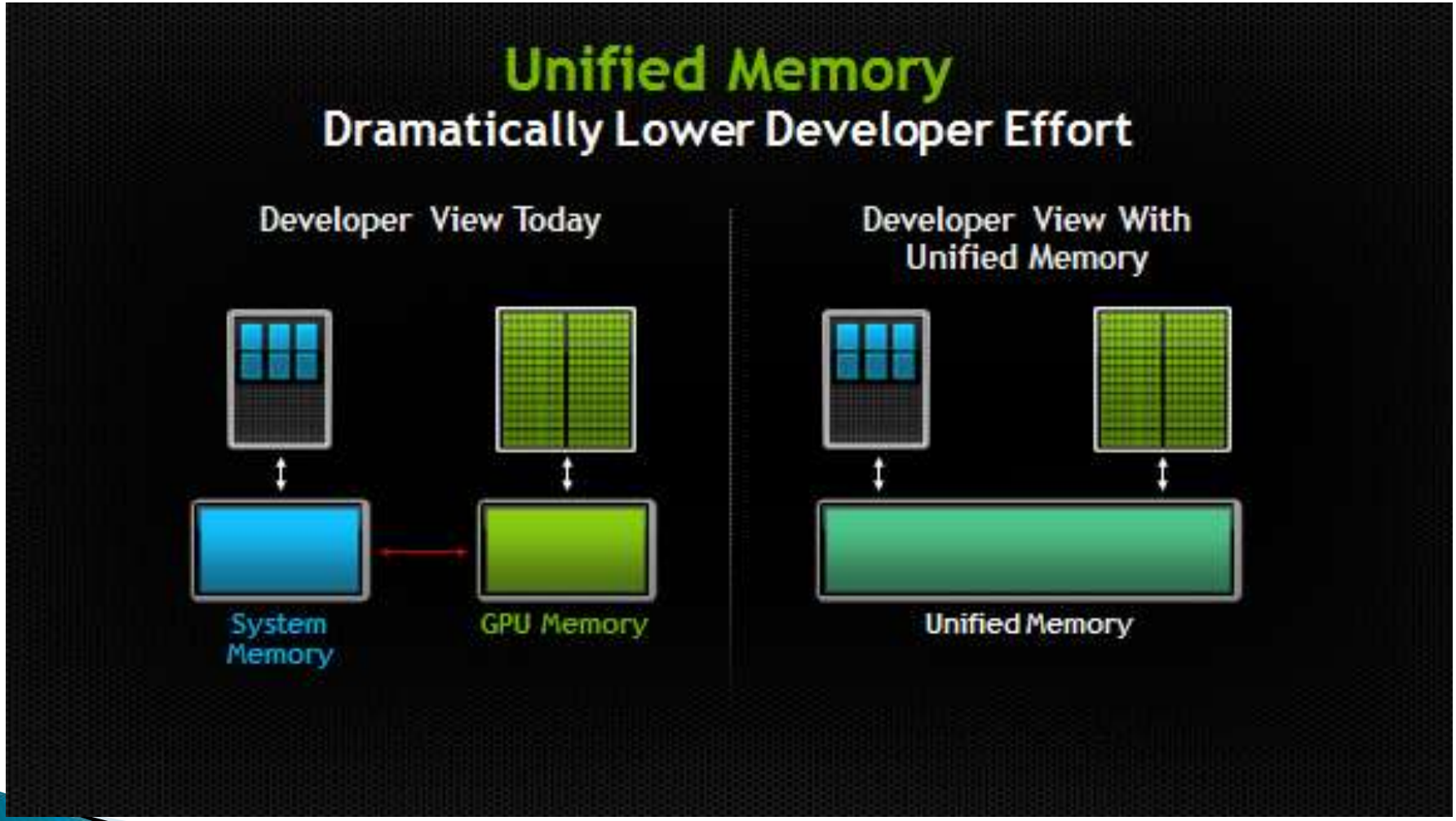
Динамически из ядер

- `malloc()` из ядра выделяет память в куче
- Не освобождается между запусками ядер
- Освобождение по `free()` только с устройства
- Доступны `memcpy()`, `memset()`

Динамически из ядер

- Память под кучу выделяется на устройстве при инициализации CUDA runtime и освобождается при завершении программы
  - После создания размер кучи не может быть изменен
  - По умолчанию – 8мб
  - Можно задать до первого вызова ядра с `malloc` через `cudaDeviceSetLimit(cudaLimitMallocHeapSize, N)`

CUDA Unified Memory



CUDA Unified Memory

- CUDA Unified Memory – это технология использования единого адресного пространства (Unified Virtual Addressing) в совокупности с автоматической синхронизацией страниц памяти
- Вообще говоря, профессионально написанная программа с ручным управлением памятью и асинхронными функциями копирования данных работает быстрее. Зато занимает больше строк кода и требует время на исправление ошибок в управлении памятью

CUDA Unified Memory

- Без Unified Memory:

```
host_a = new float[N];
host_b = new float[N];

инициализация host_a и host_b …

cudaMalloc(&dev_a, N*4);
cudaMalloc(&dev_b, N*4);
cudaMalloc(&dev_c, N*4);
cudaMemcpy(dev_a, host_a, N*4,cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, N*4,cudaMemcpyHostToDevice);

sum<<<1024, 1024>>>(dev_a, dev_b, dev_c, N);

cudaMemcpy(host_c, dev_c, N*4,cudaMemcpyDeviceToHost);
```

CUDA Unified Memory

- Unified Memory:

```
cudaMallocManaged(&a, N*4);
cudaMallocManaged(&b, N*4);
cudaMallocManaged(&c, N*4);

инициализация a и b …

sum<<<1024, 1024>>>(a, b, c, N);

cudaDeviceSynchronize();
```

CPU не может получить доступ к unified memory пока работает GPU, поэтому здесь необходим вызов cudaDeviceSynchronize()

Указатели

- Указатели использовать можно!
- Для любого типа памяти:

```
__device__ int my_global_variable;
__constant__ int my_constant_variable = 13;

__global__ void foo(void)
{
  __shared__ int my_shared_variable;

  int *ptr_to_global = &my_global_variable;
  const int *ptr_to_constant = &my_constant_variable;
  int *ptr_to_shared = &my_shared_variable;
  ...
  *ptr_to_global = *ptr_to_shared;
}
```

Указатели

- Что означает квалификатор для указателя?
  - `__shared__ int *ptr;`
- Квалификаторы относятся не к указываемой памяти, а к ячейке памяти, где хранится адрес
  - `ptr` – это разделяемая переменная-указатель, а не указатель на разделяемую память!

Не усложняйте работу компилятору!

```
__device__ int my_global_variable;
__global__ void foo(int *input)
{
  __shared__ int my_shared_variable;

  int *ptr = 0;
  if(input[threadIdx.x] % 2)
    ptr = &my_global_variable;
  else
    ptr = &my_shared_variable;
  // на какую память ссылается ptr?
}
```

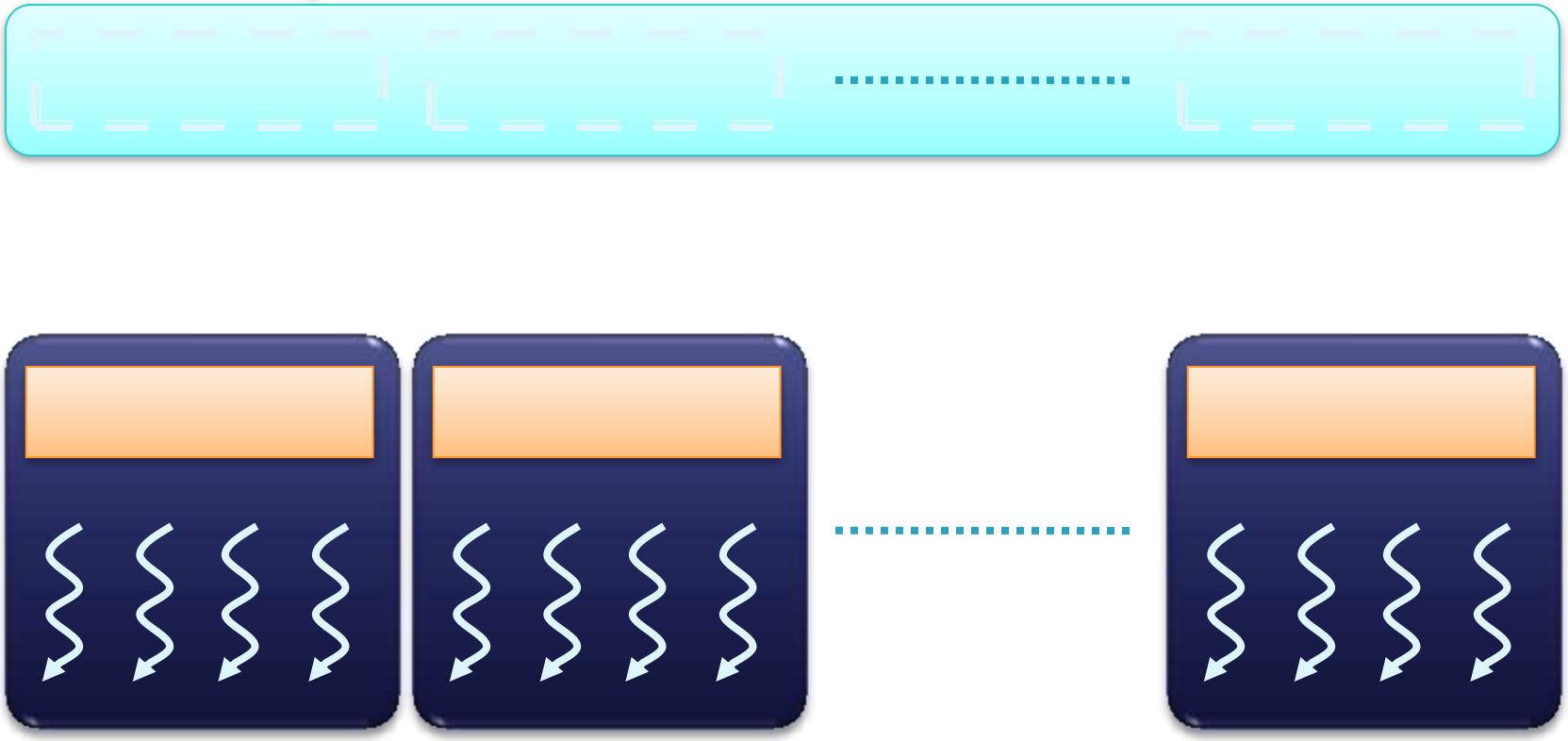
Полезные советы

- Используйте простые индексные выражения
- Избегайте копирования указателей
- Избегайте указателей на указатели
- Следите за сообщениями компилятора
  - `Warning: Cannot tell what pointer points to, assuming global memory space`

Алгоритм работы с разделяемой памятью

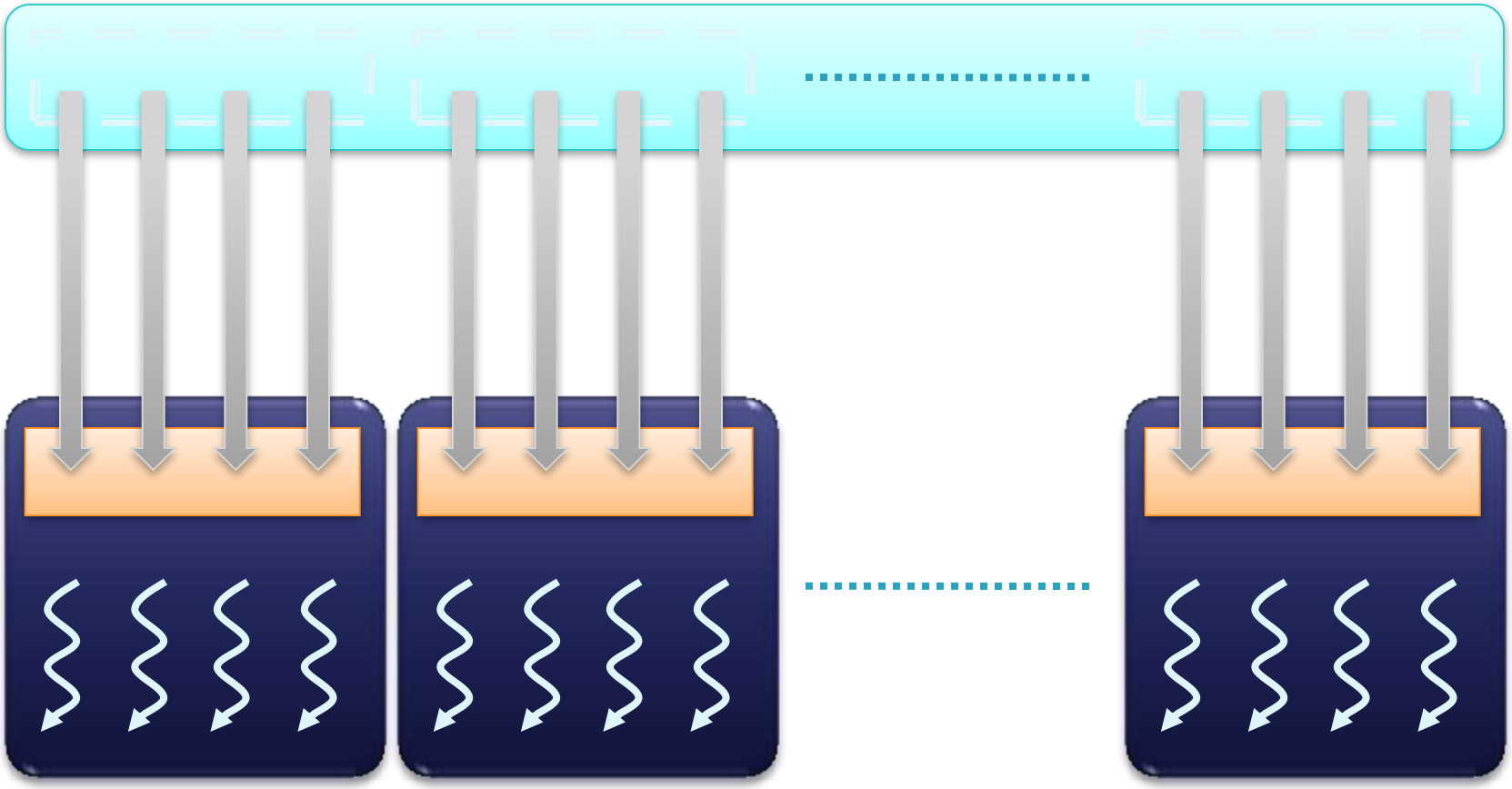
- Разделяемая память расположена на чипе и гораздо быстрее глобальной видеопамяти
- Разделите данные на блоки, которые подходят по размеру под разделяемую память каждого мультипроцессора

Алгоритм работы с разделяемой памятью



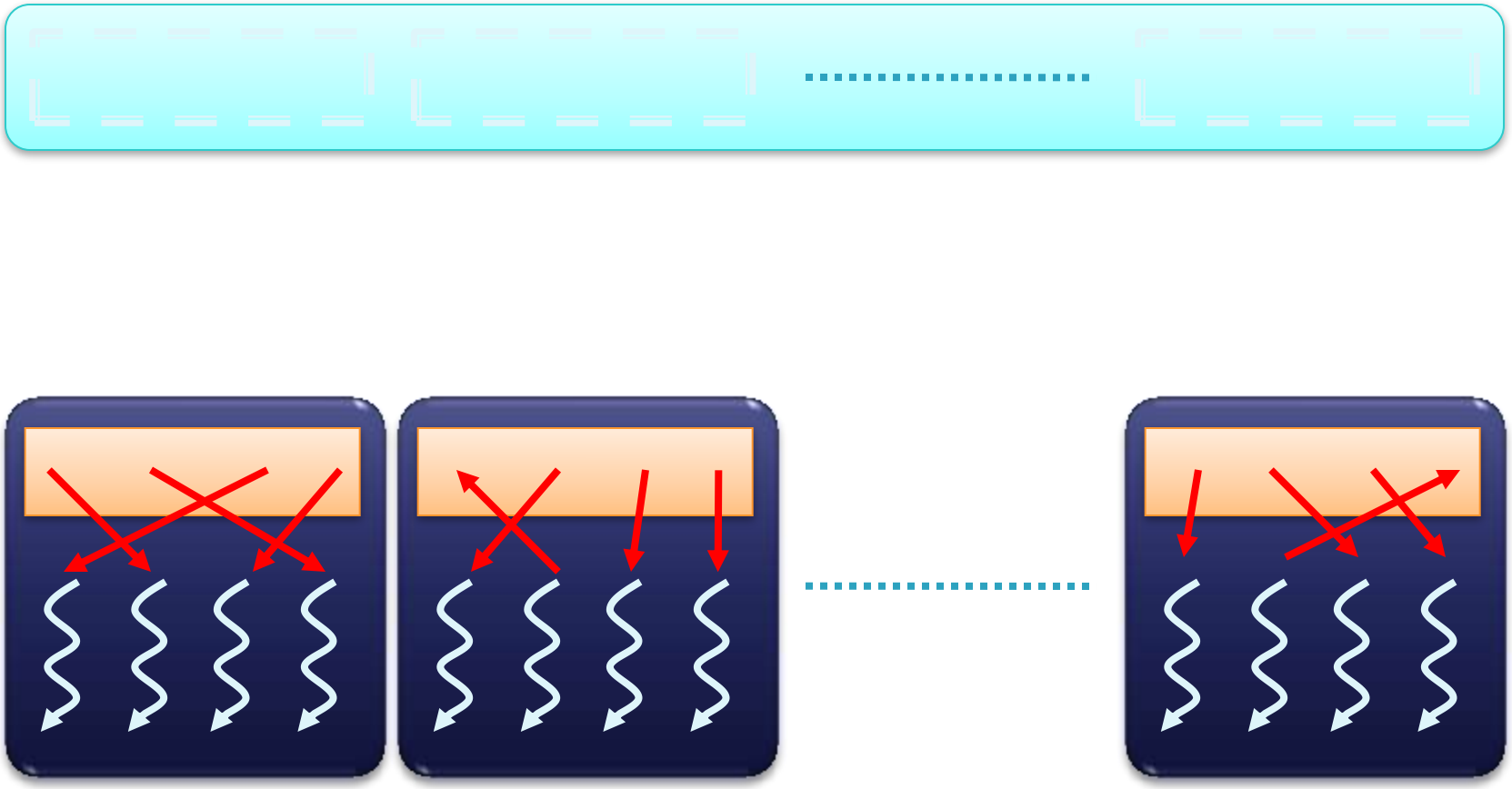
- Обработка одной порции данных должна происходить одним блоком потоков

Алгоритм работы с разделяемой памятью



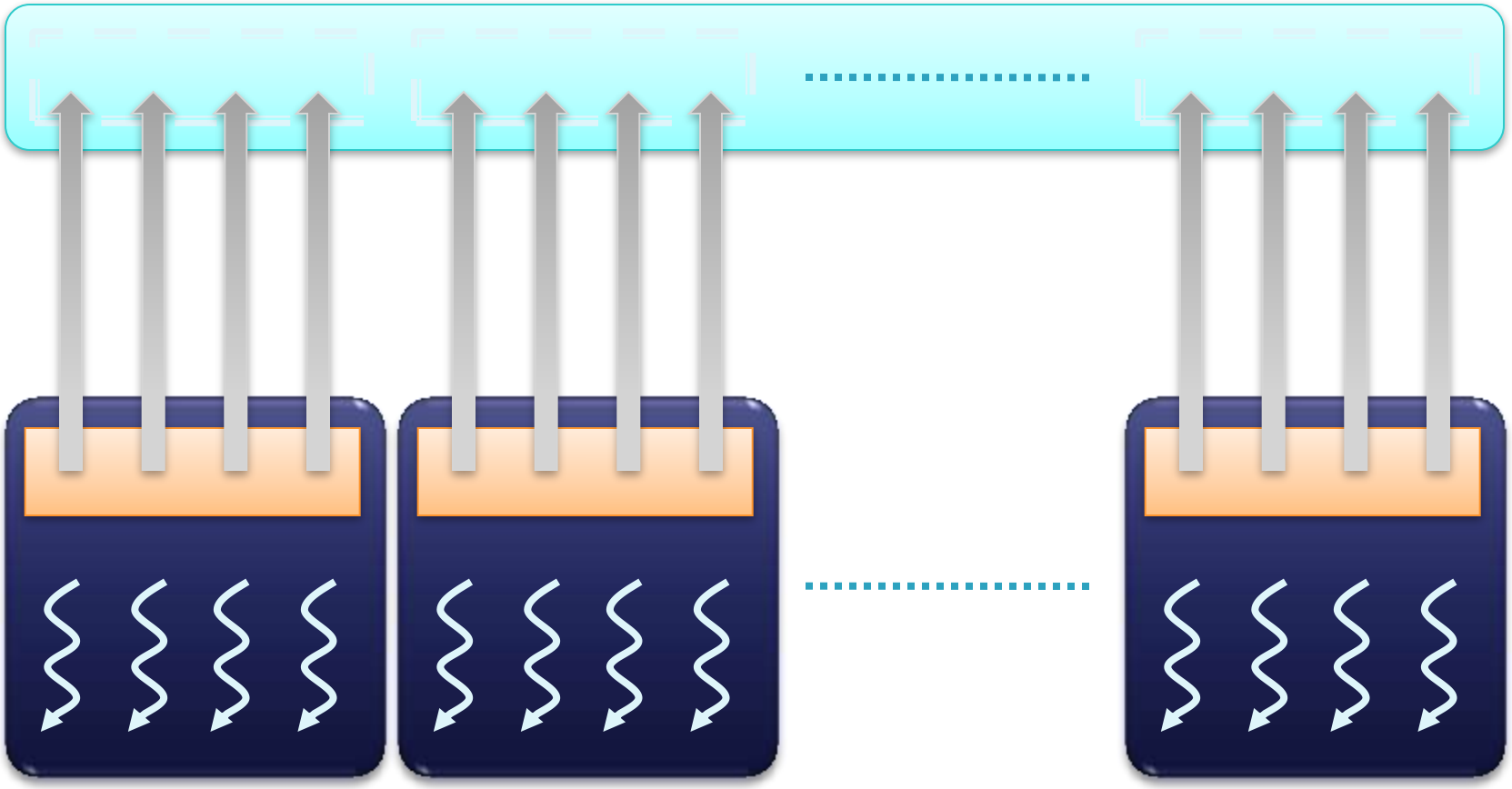
- Загружайте данные из глобальной памяти в разделяемую, используя все параллельные потоки

Алгоритм работы с разделяемой памятью



- После синхронизации произведите требуемые вычисления над данными в разделяемом кэше

Алгоритм работы с разделяемой памятью



- Скопируйте результат обратно в глобальную память

Атомарные функции

- Вопрос:

```
__global__ void race(void)
{
   __shared__ int my_shared_variable;
   my_shared_variable += 1;

   // какое значение получит
   // my_shared_variable?
}
```

Атомарные функции

- Это – состояние гонки
- Результат операции не определен
- В подобных случаях нужно использовать барьеры (`__syncthreads`) или атомарные функции

Атомарные функции

```
__global__ void race(void)
{
   __shared__ int my_shared_variable;
   atomicAdd(&my_shared_variable,1);


}
```

Список атомарных функций

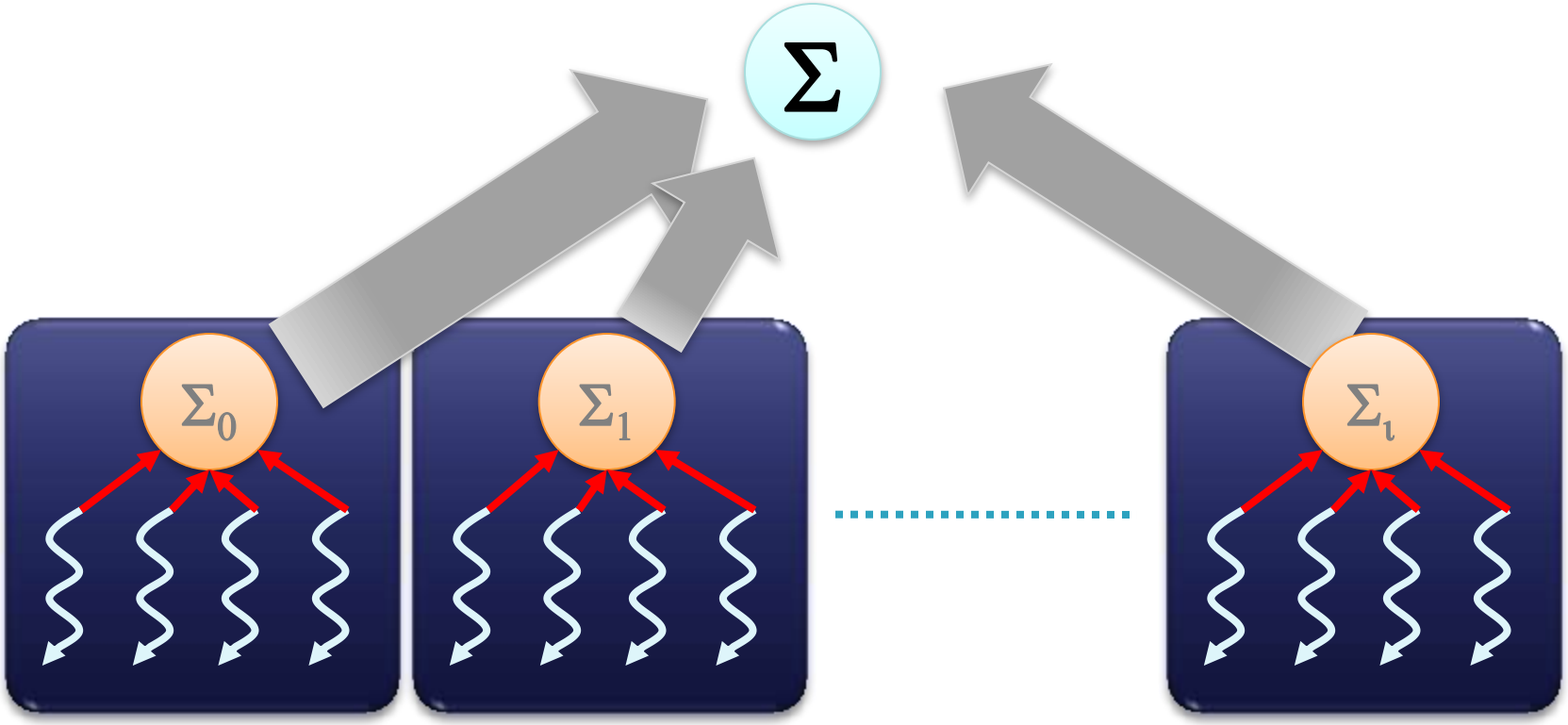
- atomicAdd, atomicSub, atomicExch, atomicInc, atomicDec, atomicMin, atomicMax, atomicCAS, atomicOr, atomicXor, atomicAnd
  - CAS – Compare and Swap (*old, compare, val):
    *old = *old==compare ? val : old; return *old
- Функции голосования (для варпа):
  __all, __any, __ballot
- Функции обмена значениями переменной (для варпа):
  __shfl, __shfl_up, __shfl_down, __shfl_xor

Производительность

- Атомарные функции работают медленно
- Они сериализуют доступ к переменной
- Пример – суммирование массива:

```
__global__ void sum(int *input, int
  *result)
{
   atomicAdd(result, input[threadIdx.x]);
}
...
// скольким потокам понадобится
// эксклюзивный доступ к result?
sum<<<B,N/B>>>(input,result);
```

Выход: использовать иерархические алгоритмы



- В каждом потоке `atomicAdd` для разделяемой переменной частичной суммы
- В каждом блоке `atomicAdd` для общей переменной

Иерархический алгоритм

```
__global__ void sum(int *input, int
  *result)
{
   __shared__ int partial_sum;

   // thread 0 is responsible for
   // initializing partial_sum
   if(threadIdx.x == 0)
     partial_sum = 0;
   __syncthreads();

   ...
}
```

Иерархический алгоритм

```
__global__ void sum(int *input, int *result)
{
  ...
  id = threadIdx.x + blockIdx.x*blockDim.x;
  // each thread updates the partial sum
  atomicAdd(&partial_sum, input[id]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0)
    atomicAdd(result, partial_sum);
}
```

Задача 1

A – двумерная матрица m x n, хранящаяся в памяти по строкам. Напишите программу, в которой каждый поток выполняет одно присваивание:

Aij = f(i,j), используя

а) двумерную нумерацию потоков и блоков;

б) одномерную

в) смешанную

Задача 2

A – двумерная матрица m x n, хранящаяся в памяти по строкам. Напишите кусочек программы, в котором каждый блок потоков копирует в разделяемую память

а) всю строчку матрицы

б) часть строки матрицы (т.к. вся строка не помещается)

Грид должен быть таким, чтобы была обработана вся матрица.