Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

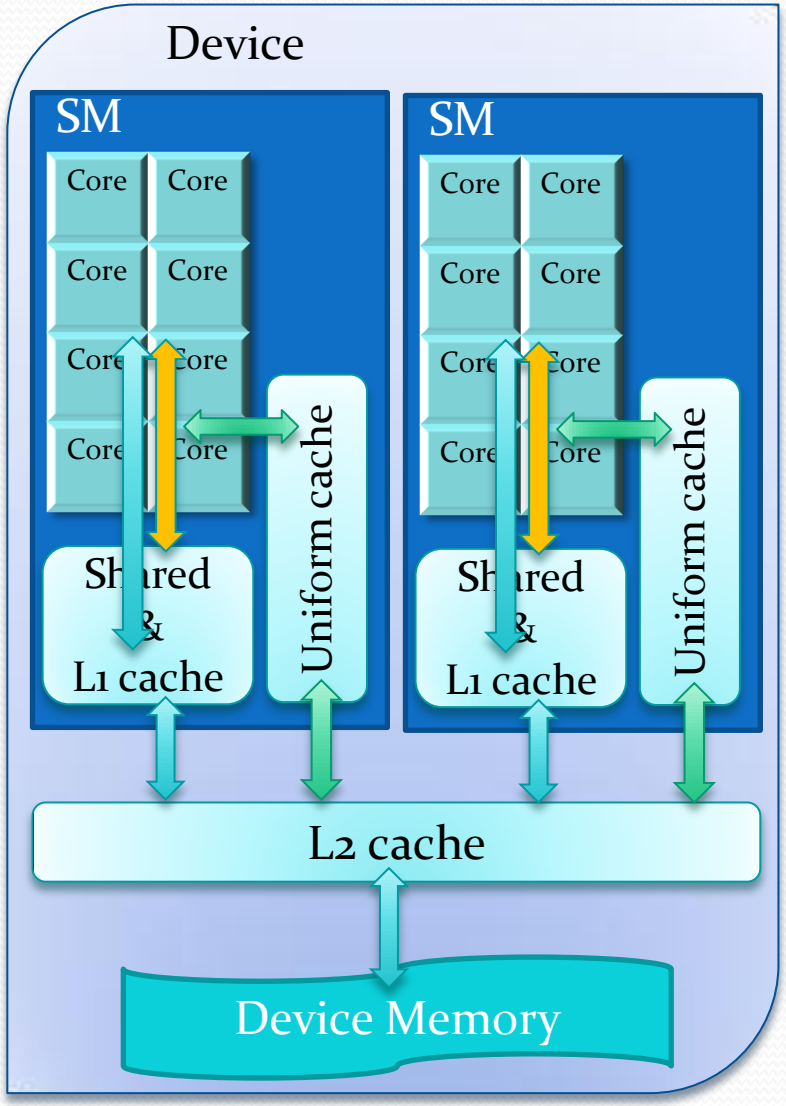
Лихогруд Николай
n.lihogrud@gmail.com

Часть четвертая

Константная память

Константная память

- Расположена в **DRAM GPU**
- Объём до 64KB
  - Параметр устройства **totalConstMem**
- **Кешируется** в специальном read-only кеше – Unifrom Cache
  - Объём 8 KB

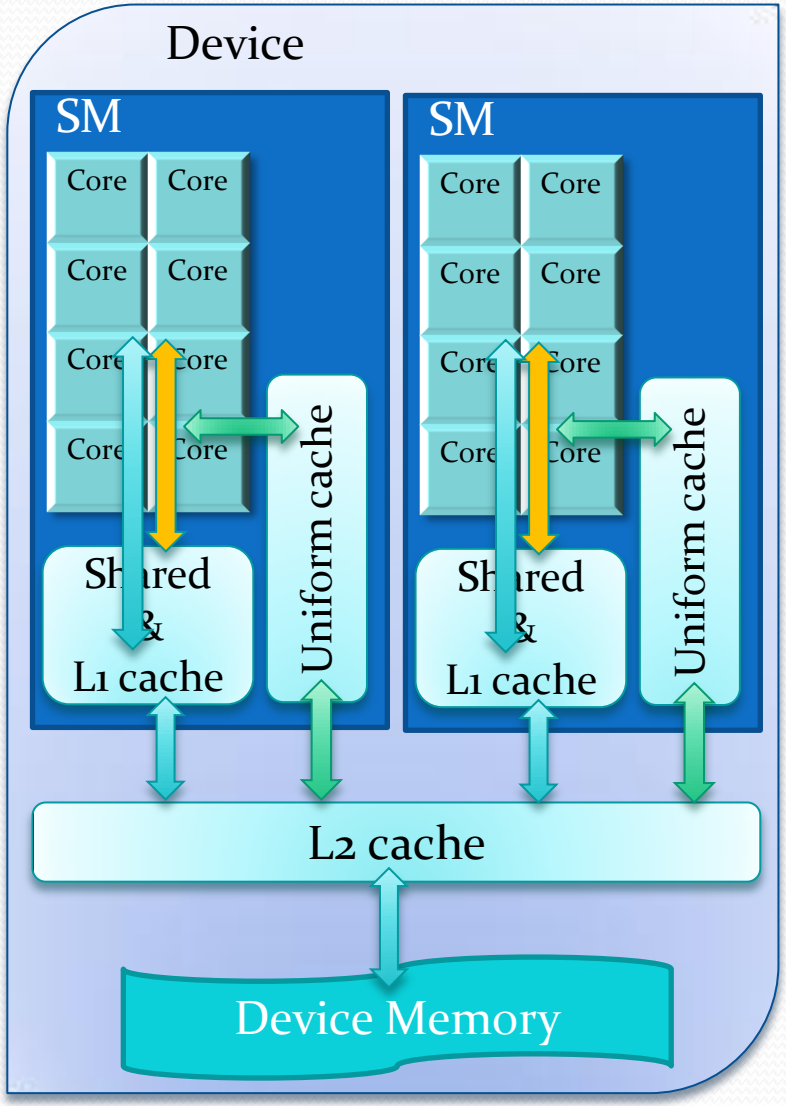


Константная память

Возможные обмены
между устройствами
при обработке обращений в
глобальную память

Возможные обмены
между устройствами
при обработке обращений
в общую память

Возможные обмены
между устройствами
при обработке обращений
в константную память



Объявление

- В глобальной области видимости

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

- Можно ещё дополнительно указать __device__ , чтобы подчеркнуть, что память выделяется на устройстве :

```
__device__ __constant__ int constVar2;
```

Особенность __device__ и __constant__

- Переменные с атрибутами `__device__` и `__constant__` находятся в глобальной области видимости и хранятся в объектном модуле как отдельные символы
- Память под них выделяется на устройстве автоматически при старте приложения, освобождается при завершении
- Работать с ними на хосте можно через функции `cudaMemcpyToSymbol() , cudaMemcpyToSymbolAsync(), cudaGetSymbolAddress(), cudaMemcpyFromSymbol(), cudaMemcpyFromSymbolAsync(), cudaGetSymbolSize()`

Особенность __constant__

- Доступна **на чтение (и только на чтение!)** из любой нити любого грида обычным способом:

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

Пример

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

- На хосте:

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

Обращения в константную память

- Обращение выполняется одновременно для всех нитей варпа (SIMT)
- Исходное обращение разбивается на столько запросов, сколько различных адресов в нём было
  - Каждый запрос выполняется либо через запрос к кешу в случае кеш-попадания, либо через глобальную память
  - Если их было n, то пропускная способность уменьшается в n раз

Обращения в константную память

```
__constant__ int constMem[32];
__global__ void kernel() {
    …
    int a = constMem[ threadIdx.x / 32 ]; // 1 запрос в
                                          константную
                                          память
    int a = constMem[ threadIdx.x]; // 32 запроса в
                                       константную
                                       память
    …
}
```

Однородные обращения

- Помимо обработки запросов в константную память, Uniform Cache обрабатывает «Однородные» обращения (Unifrom Accesses) – когда все нити варпа обращаются в глобальную память по одному адресу
  - При выполнении требований:
    - Доступ только по чтению
    - Адрес не зависит от индекса нити в блоке (`threadIdx`)

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

Компилятор может заменить в ассемблере обычную инструкцию загрузки из глобальной памяти на инструкцию однородной загрузки, которая будет выполнена через **Uniform Cache**

Однородные обращения

- При выполнении требований:
  - Доступ только по чтению
  - Адрес не зависит от индекса нити в блоке (`threaIdx`)

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

- Второе требование гарантирует, что все нити варпа обращаются по одному адресу

- Чтобы помочь компилятору с первым требованием, можно пометить указатели атрибутом `const __restrict`

Передача параметров в ядра

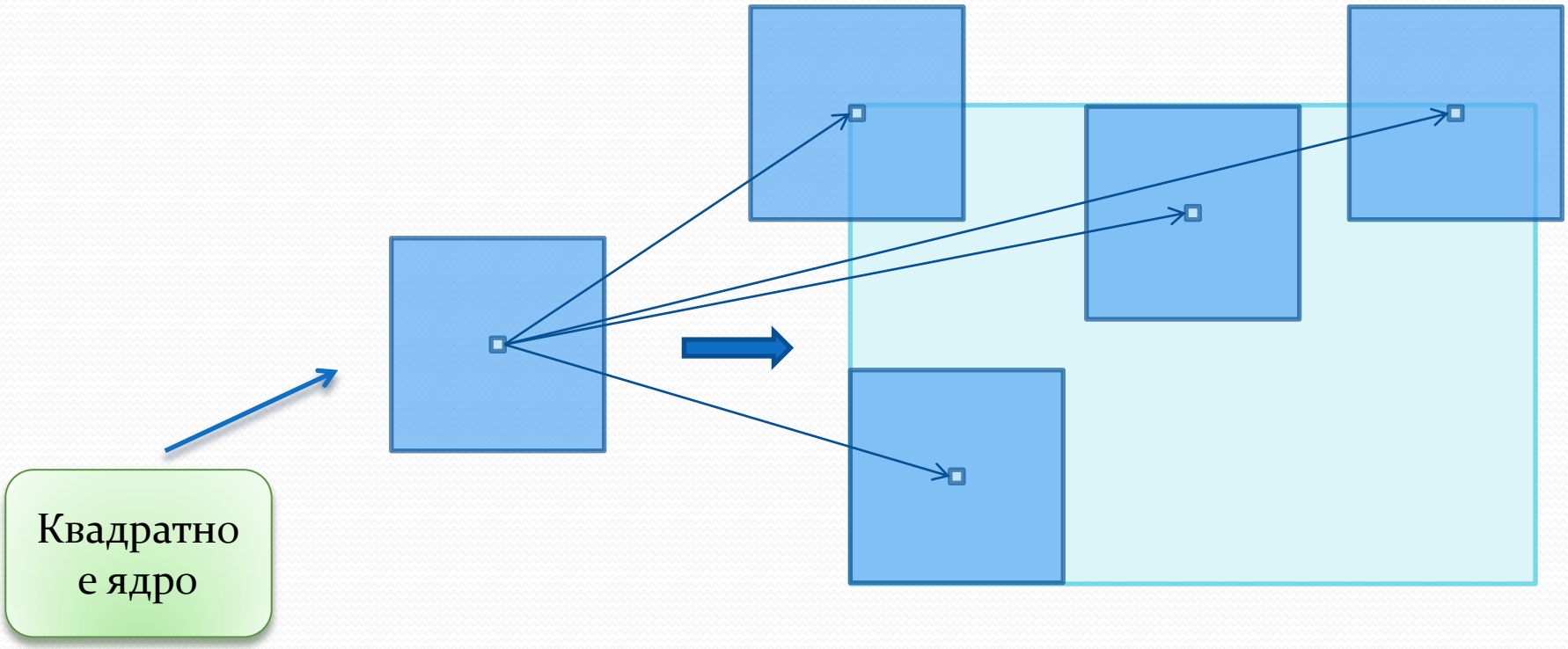
- Параметры передаются в ядра через константную память
  - Параметры передаются в единственном экземпляре для всех нитей грида
  - Это приемлимо, т.к.,
    - в основном, нити варпа обращаются к одному и тому же параметру -> Uniform Access
    - После первого варпа параметры уже будут в кеше
- Суммарный размер передаваемых параметров долен быть **не больше, чем 4 КВ**

Передача грида в ядра

- Помимо параметров, через константную память передаются размеры грида: `gridDim, blockDim`
  - `threadIdx, blockIdx` нить получает из спец. регистров (заведомо не Uniform)
  - `gridDim, blockDim` нить **считывает из константной памяти** в самом начале работы (Uniform)

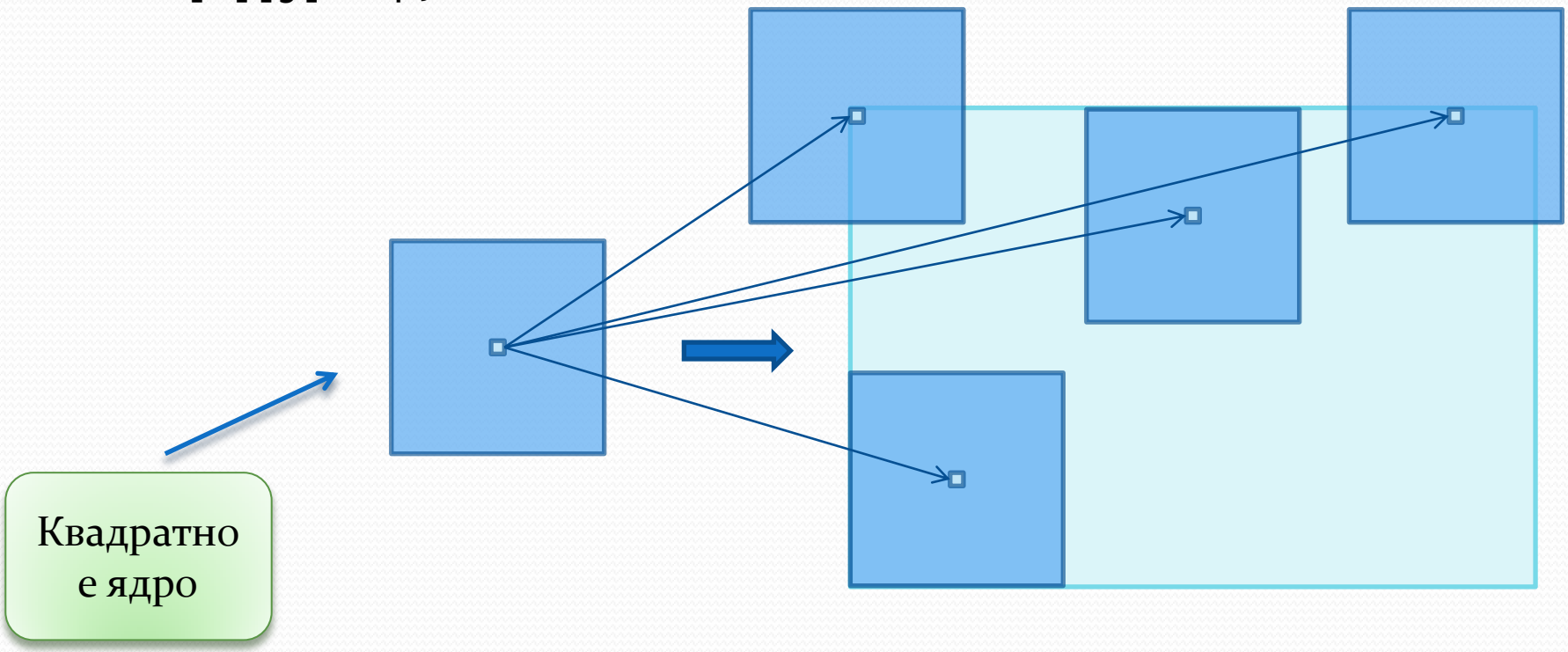
Пример: фильтрация с ядром

- Наложить ядро на окрестность каждого элемента
  - Для элемента (i,j) записать в (i, j) матрицы-результата сумму произведений элементов окрестности на соответствующие коэффициенты фильтра



Пример: фильтрация с ядром

```
tmp = 0;
for(ik=0..2*r)
    for(jk=0..2*r)
     tmp += matrix[i+ik-r][j+jk-r]*kernel[ik][jk];
result[i][j]=tmp;
```



Пример: фильтрация с ядром

```
tmp = 0;
for(ik=0..2*r)
    for(jk=0..2*r)
     tmp += matrix[i+ik-r][j+jk-r] * kernel[ik][jk];
result[i][j]=tmp;
```

Не зависит от i,j
Все нити варпа читают один элемент
Можно расположить в константной памяти

Пример: преобразование координат

- Поворот, перенос, сжатие/растяжение объекта в пространстве выполняются путем умножения всех вершин объекта на матрицу преобразования координат

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{31} & m_{33} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix},$$

$[x \;\; y \;\; z \;\; w]^T$ - однородные координаты вершины

$[x \;\; y \;\; z \;\; 1]^T$ - нормированные однородные координаты вершины

$[x' \;\; y' \;\; z' \;\; w']^T$ - координаты вершины после преобразования

Пример: преобразование координат

- При реализации на CUDA каждая нить умножает координаты одной вершины на матрицу преобразования координат

$$id \ - \text{номер нити}$$
$$vertexes \ - \text{массив вершин}$$
$$M \ - \text{матрица преобразования координат}$$

$$vertexes[id] = M \times vertexes[id]$$

- Матрица преобразования координат для всех нитей одна, доступ к ней только на чтение, нити варпа обращаются к одному элементу -> Можем разместить её в константной памяти

Пример: преобразование координат

- Вспомогательные типы для упрощения ядра:

```
typedef float CoordinateType;
typedef CoordinateType TransformMatrixRow[4]; // строка матрицы преобразования
typedef TransformMatrixRow TransformMatrix[4]; // матрица преобразования как 4
                                                  строки

// вершина фигуры
struct Vertex {
    CoordinateType x,y,z,w; // однородные координаты
    __device__ Vertex(CoordinateType x = 0, CoordinateType y = 0,
                      CoordinateType z = 0, CoordinateType w = 1)
               :x(x), y(y), z(z), w(w) {} // конструктор
    __device__ void normalize() {…} // нормализация – разделить x, y, z на w

    // умножение на строку матрицы преобразования
    __device__ CoordinateType operator*(TransformMatrixRow row) {
        return x * row[0] + y * row[1] + z * row[2] + w * row[3];
    }
};
```

Пример: преобразование координат

- Вспомогательные типы для упрощения ядра:

```
// вершины фигуры
struct Vertexes {

    CoordinateType *x, *y, *z; // структура с массивами, а не массив структур
    int count; // число вершин

    // получение отдельной вершины
    __device__ Vertex operator[](int n) {
         return Vertex(x[n], y[n], z[n]);
    }

    // запись вершины
    __device__ void setVertexAtIndex(Vertex vertex, int index) {
        x[index] = vertex.x;
        y[index] = vertex.y;
        z[index] = vertex.z;
    }
};
```

Пример: преобразование координат

- Матрица преобразования и ядро:

```
__constant__ TransformMatrix transformMatrix; // матрица преобразования в
                                                 константной памяти

__global__ void coordinateTransform(Vertexes vertexes) {

    int id = blockIdx.x * blockDim.x + threadIdx.x; // индекс нити
    if (id >= vertexes.count) { // проверка на выход за границу
        return;
    }
    Vertex vertex = vertexes[id]; // считали вершину
    Vertex newVertex( vertex * transformMatrix[0],
                      vertex * transformMatrix[1],
                      vertex * transformMatrix[2],
                      vertex * transformMatrix[3]); // умножение на матрицу
    newVertex.normalize(); // нормализация
    vertexes.setVertexAtIndex(newVertex, id); // запись результата
}
```

Пример: преобразование координат

- Результаты тестов:
  - Tesla K20c, 65 миллионов вершин

	Точность	
Атрибут матрицы	Float	Double
__constant__	12.392ms	22.161ms
__device__	50.546ms	53.667ms

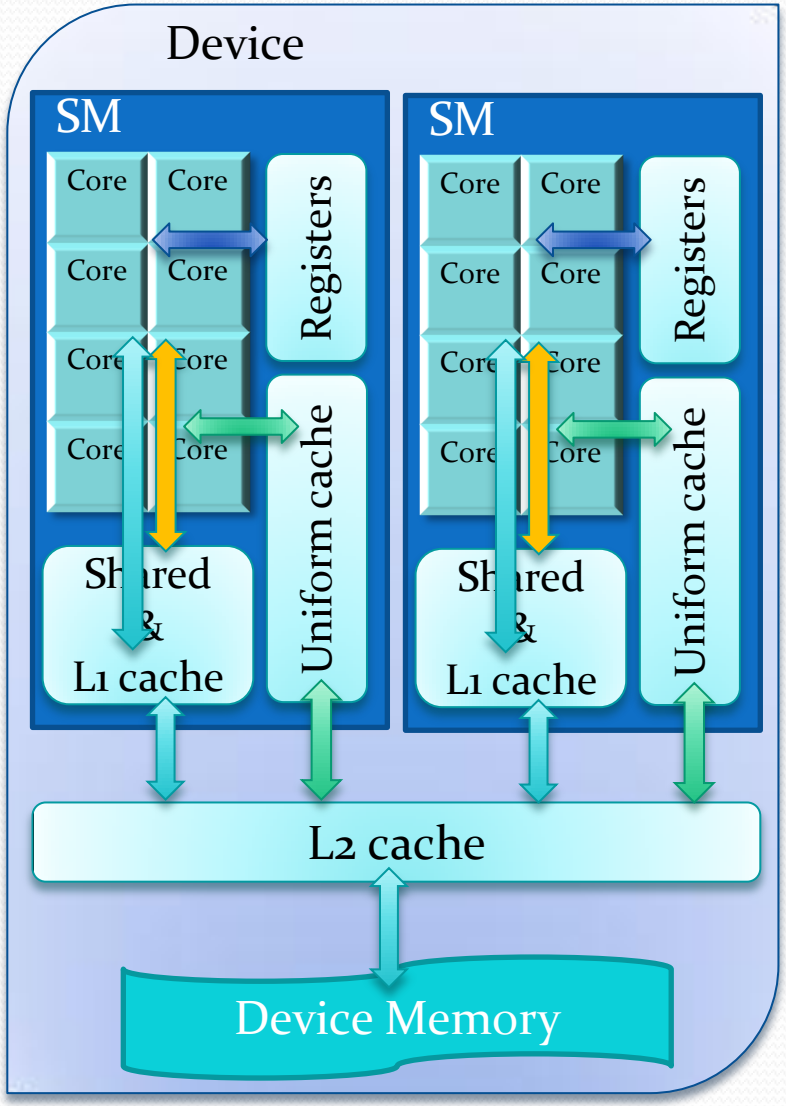
Выводы

- Использование константной памяти:
  - Позволяет разгрузить кеш L1
  - Ускоряет выполнение при выполнении требования однородности обращений

Регистры и локальная память

Регистровая память

- Расположена на мультипроцессоре
- Самая быстрая память
- Недоступна явно в программе
  - Инструкции с использованием регистров генерируются компилятором
- Каждая нить является эксклюзивным пользователем своих регистров на всё время жизни
  - Регистры делятся между всеми резидентными нитями на мультипроцессоре
  - Быстрое переключение контекста



Регистры в Fermi и Kepler

- Параметр устройства `regsPerMultiprocessor` - число 32-битных регистров на каждом мультипроцессоре. Разделяются межу виртуальными блоками, работающими на мультипроцессоре.

- Для Fermi:
  - Каждый мультипроцессор содержит **32768** 32-битных регистров (128 КВ)
  - Отдельной нити доступно максимум **63** регистра
  - Максимальное число регистров при occupancy = 1: **32768** / **1536** = **21**

- Для Kepler:
  - Каждый мультипроцессор содержит **65536** 32-битных регистров (256КВ)
  - Отдельной нити доступно максимум **255** регистров
  - Максимальное число регистров при occupancy = 1: **65536** / **2048** = **32**

Особенность выделения регистров

- Регистры выделяются пачками отдельно для каждого варпа
  - Пусть размер пачки - ***regAllocUnitSize***
  - Пусть каждой нити нужно ***nRegs*** регистров. Тогда общее число регистров, необходимых блоку из ***blockSize*** нитей:

$$\left\lceil \frac{nRegs * 32}{regAllocUnitSize} \right\rceil * regAllocUnitSize * \left\lceil \frac{blockSize}{32} \right\rceil$$

Число пачек на варп

Число варпов в блоке

⌈ ⌉ – ближайшее сверху целое

Особенность выделения регистров

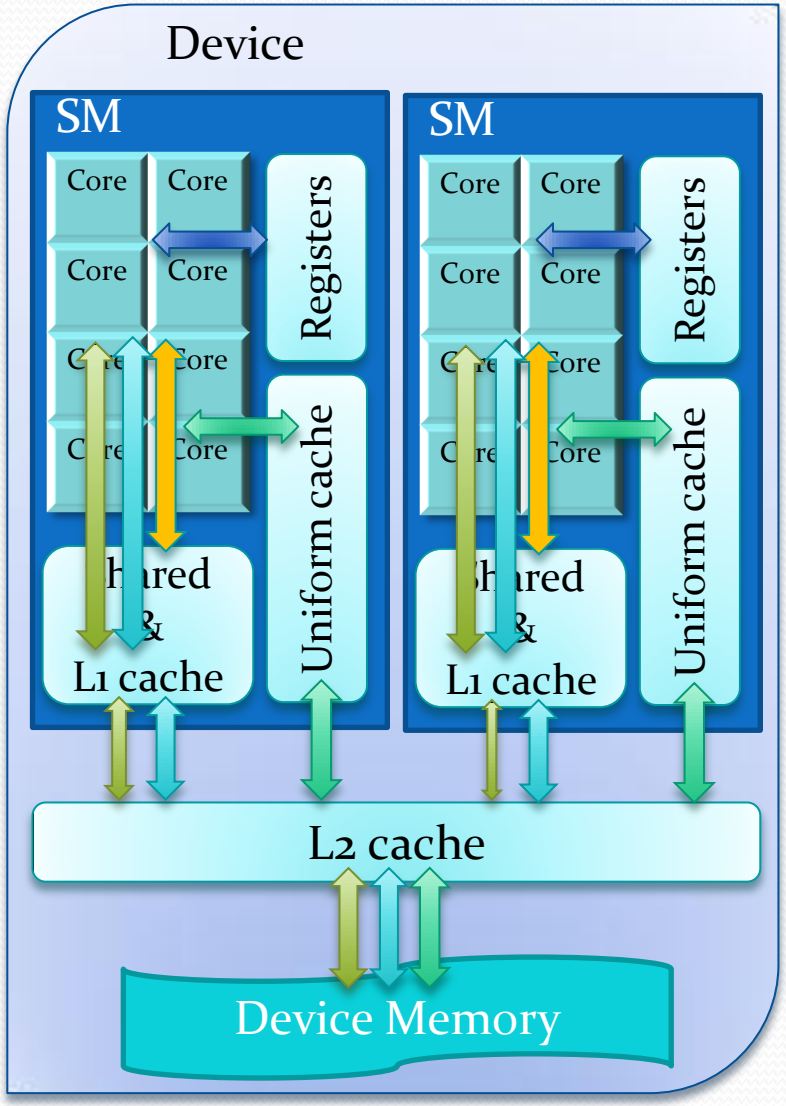
$$\left\lceil \frac{nRegs * 32}{regAllocUnitSize} \right\rceil * regAllocUnitSize * \left\lceil \frac{blockSize}{32} \right\rceil$$

- Для **Fermi** $regAllocUnitSize = 64$
  - Пусть nRegs = 21 а блок – 512 нитей(16 варпов):
    - ✓ $\left\lceil \frac{21*32}{64} \right\rceil = 11$ пачек, $11 * 64 * 16 = 11264$ - столько регистров выделится одному блоку
    - ✓ $11264 * 3 = 33792 > 32768$, т.е. на три блока регистров не хватит
  - Получаем, что максимальное число регистров при occupancy=1 и блоке из 512 нитей равно 20

- Для **Kepler** $regAllocUnitSize = 256$

Локальная память

- Расположена в **DRAM**
  - Высокая латентность
- Доступ осуществляется по тем же правилам, что и запросы в глобальную память
  - Кеширование в L1
  - Транзакции
- Недоступна явно в программе
  - Инструкции обращения в локальную память генерируются компилятором
- Обладает упрощённой схемой адресации
  - Оптимизирована для минимизации количества транзакций



Когда используется локальная память?

- Обычно компилятор помещает на регистры все локальные переменные
- Но есть исключения, размещаемые в локальной памяти:
  - Массивы, для которые не всегда можно определить к какому элементу в какой момент времени идёт доступ (не константные индексы)
  - Большие массивы или структуры, которые использовали бы слишком много регистров
  - Любая переменная, если превышен лимит регистров на нить (так называемый «спиллинг регистров» register spilling)

Когда используется локальная память?

- Некоторые встроенные математические функции могут использовать локальную память
- Через локальную память передаётся часть операндов при вызове функций
  - В т.ч. в локальной памяти моделируется стек фреймов при рекурсивных вызовах

Пример

```
__device__ int deviceFunc(int *a) {
      int x; // скорее всего на регистре (если нет спиллинга)
      int array[10]; // может в локальной, т.к. далее этот массив
                индексируется неизвестным при компиляции индексом
      ...
      x = sinf(threadIdx.x) // sinf может
        использовать локальную память
      x = x + array[ threadIdx.x % 10 ];
      if (x < 100) {
          x = x + deviceFunc(a); // фрейм вызова будет
              // сохранен в стеке, расположенном в локальной памяти
    }
    return x;
}
```

nvcc -Xptxas -v

- Выводит количество регистров, константной памяти, локальной памяти и статической общей памяти, используемые ядром:

```
$:~/programming/testMod$ 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]
```

Статические ресурсы и occupancy

- Факторы, влияющие на occupancy:
  - Не более 1536 нитей на sm, не более 8 блоков
  - Не более 48KB общей памяти на sm
  - 32768 регистров на sm

Пусть ядро использует 63 регистра и размер блока 384 нити
32768 / 63 = 520 – максимум нитей

- на sm будет работать всего один блок из 384 нитей (occupancy = 0.25)

Статические ресурсы и occupancy

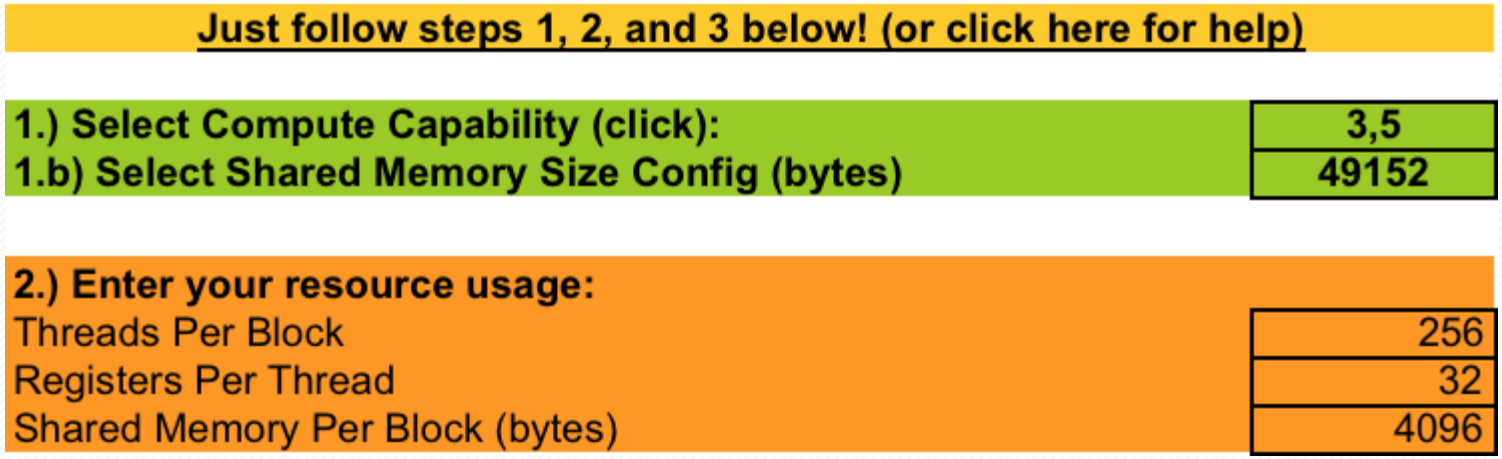
- Факторы, влияющие на occupancy:
  - Не более 1536 нитей на sm, не более 8 блоков
  - Не более 48KB общей памяти на sm
  - 32768 регистров на sm

Пусть на блок из 512 нитей нужно 32KB общей памяти

- на sm будет работать всего один блок из 512 нитей (occupancy = 0.33)

CUDA occupancy calculator

- http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



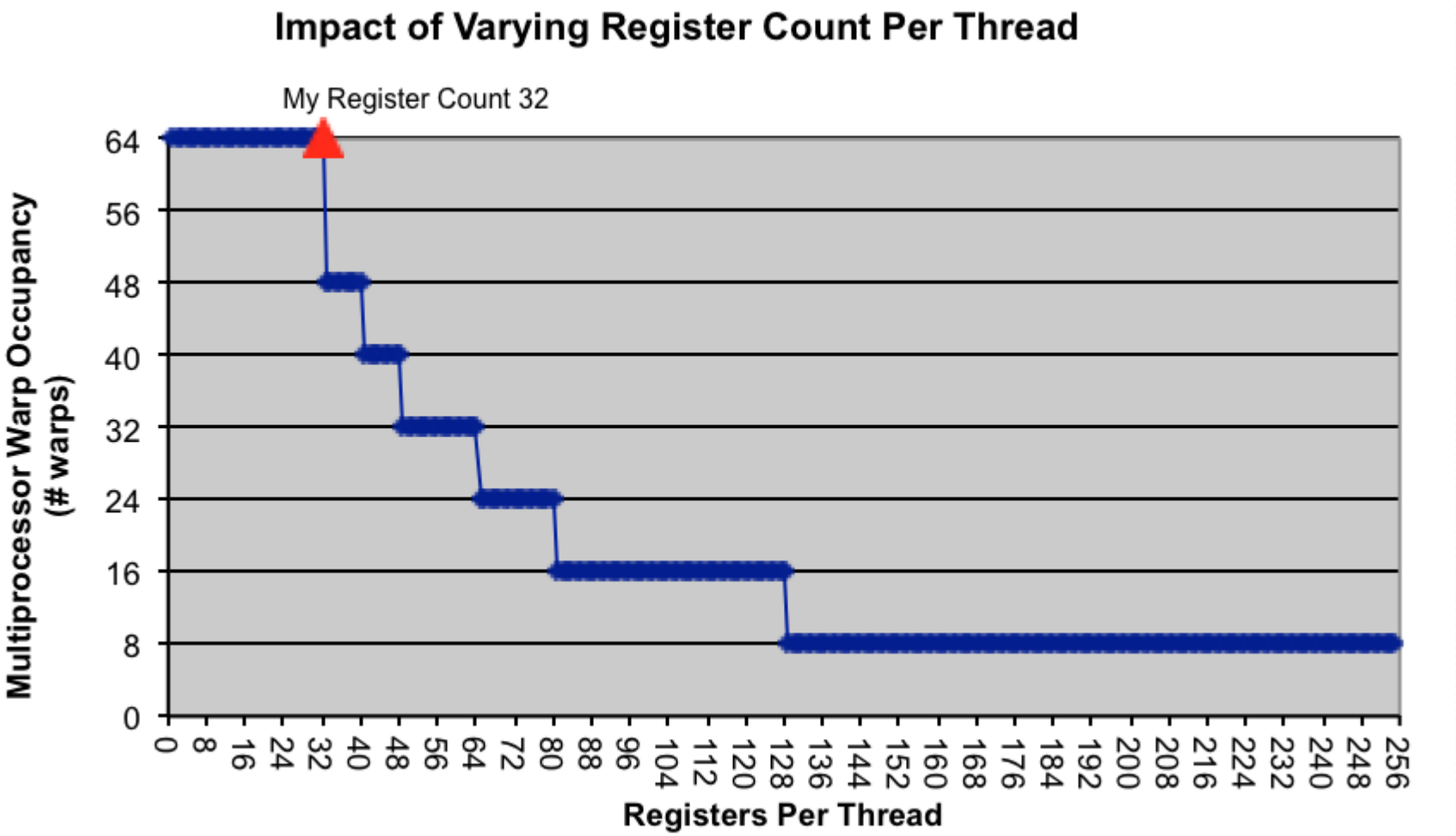
Just follow steps 1, 2, and 3 below! (or click here for help)	
1.) Select Compute Capability (click):	3,5
1.b) Select Shared Memory Size Config (bytes)	49152
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	32
Shared Memory Per Block (bytes)	4096

CUDA occupancy calculator

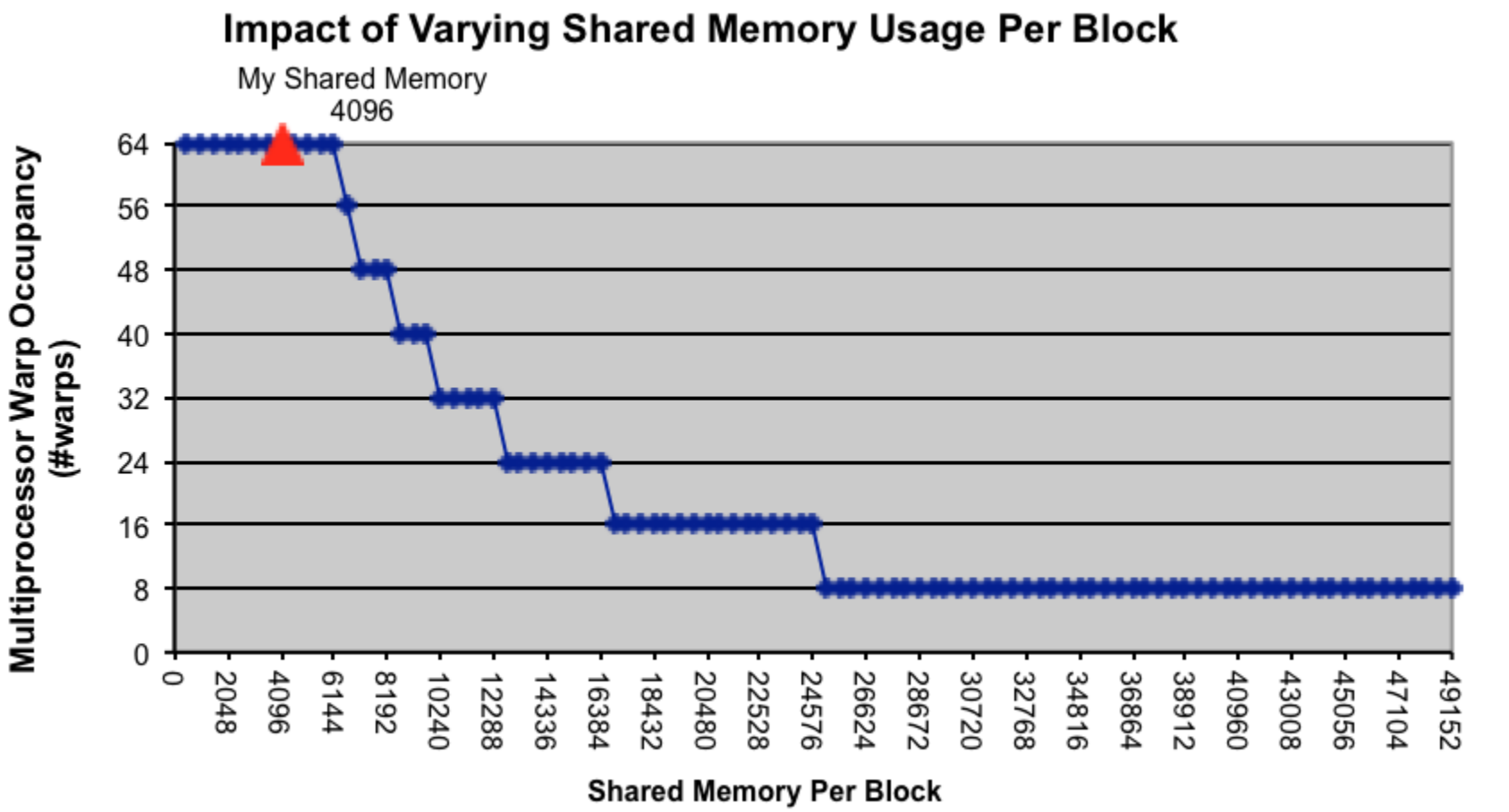
- http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

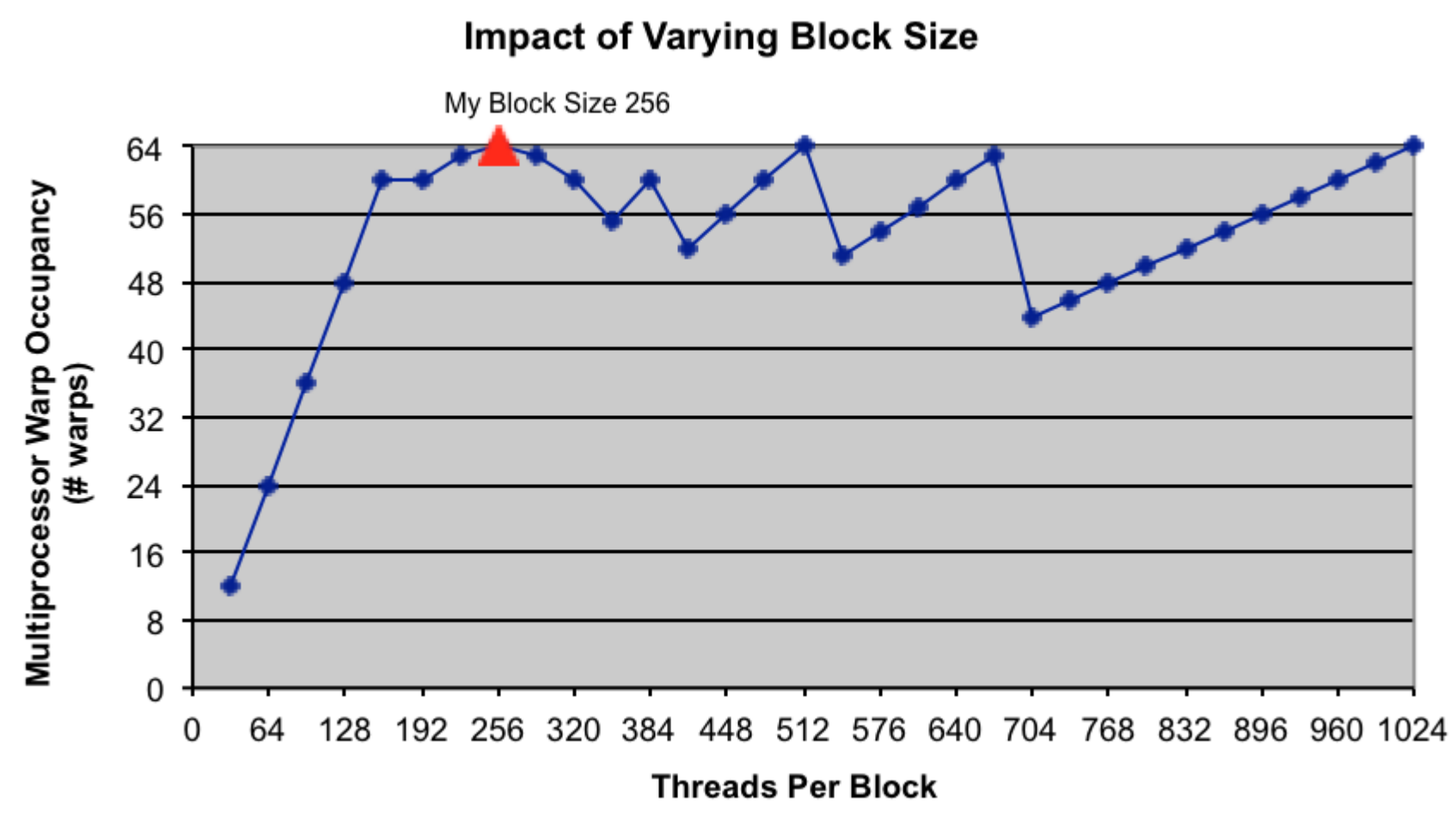
CUDA occupancy calculator



CUDA occupancy calculator



CUDA occupancy calculator



Нехватка ресурсов для запуска

- Если на мультрипроцессоре не хватает регистров даже для одного блока, то произойдет ошибка запуска
  - Нужно уменьшить размер блока
- Пример:
  - Ядро, потребляющее 37 регистров, запустится на блоке из 1024 нитей на **Kepler** и не запустится на **Fermi**
- То же самое и в отношении общей памяти – для успешного запуска её должно хватать хотя бы на один блок

Ограничение числа регистров

```
$nvcc –maxrregcount 20 –c –arch=sm_20 kernel.cu
```

- При компиляции принудительно ограничить максимальное число используемых регистров
  - Можно контролировать occupancy
  - Приводит к спиллингу регистров

Оганичение числа регистров

Получаем использование локальной памяти:

```
$ nvcc -arch=sm_20  -Xptxas -v -c kernel.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6matmul14cudaPitchedPtrS_S_' for 'sm_20'
ptxas info    : Function properties for _Z6matmul14cudaPitchedPtrS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 128 bytes cmem[0]

$ nvcc -arch=sm_20 -maxrregcount 21 -Xptxas -v -c kernel.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6matmul14cudaPitchedPtrS_S_' for 'sm_20'
ptxas info    : Function properties for  Z6matmul14cudaPitchedPtrS_S
    96 bytes stack frame, 132 bytes spill stores, 112 bytes spill loads
ptxas info    : Used 21 registers, 128 bytes cmem[0]
```

Выводы

- Всегда компилировать с **`-Xptxas -v`** и **`-arch=sm_??`**
  - **`-Xptxas -v`** выводит использование ресурсов только для архитектуры, для которой происходит компиляция
- Учитывать потребление регистров и общей памяти при задании грида для максимизации occupancy
  - Использовать для этого **occupancy calculator**
- Минимизировать использование локальной памяти
  - Надежный способ узнать из-за чего возникло использование локальной памяти – посмотреть ptx/ассемблер, об этом в другой части

The end