



Вопросы программирования и оптимизации приложений на CUDA.

⌘ Лекторы:

☑ Обухов А.Н. (Nvidia)

☑ Боресков А.В. (ВМиК МГУ)

☑ Харламов А.А. (Nvidia)

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
- ⌘ Паттерны программирования на CUDA
- ⌘ Разное

Содержание



⌘ Процесс разработки программ CUDA

- ☐ Портирование части приложения

- ☐ Общие рекомендации по оптимизации

- ☐ Инструментарий

⌘ Работа с различными типами памяти

⌘ Паттерны программирования на CUDA

⌘ Разное

Процесс разработки программ CUDA

Портирование части приложения

⌘ Определение класса портируемой задачи

- ☑ Уровень параллелизма. SIMD
- ☑ Классы задач, которые в общем случае невозможно распараллелить:
 - ☒ сжатие данных
 - ☒ IIR-фильтры
 - ☒ другие рекурсивные алгоритмы



Процесс разработки программ CUDA

Портирование части приложения

1. Оригинальный код

```
for (int i=0; i<maxIter; i++)
{
    int SAD = SAD64(data1 + i * 64, data2 + i * 64);
    processDataChunkDivergent(data1 + i * 64, SAD);
}
```

2. Вынос портируемой части из-под цикла

```
int SADvals[maxIter];
for (int i=0; i<maxIter; i++)
{
    SADvals[i] = SAD64(data1 + i * 64, data2 + i * 64);
}
for (int i=0; i<maxIter; i++)
{
    int SAD = SADvals[i];
    processDataChunkDivergent(data1 + i * 64, SAD);
}
```

3. Портирование вынесенной части на CUDA

```
cudaMemcpy(devdata1, data1, maxIter * 64, cudaMemcpyHostToDevice);
cudaMemcpy(devdata2, data2, maxIter * 64, cudaMemcpyHostToDevice);
SAD64_CUDA<<<maxIter, 64>>>(devdata1, devdata2, devSADvals);
cudaMemcpy(SADvals, devSADvals, maxIter, cudaMemcpyDeviceToHost);
```

Содержание



⌘ Процесс разработки программ CUDA

- ☐ Портирование части приложения

- ☐ Общие рекомендации по оптимизации

- ☐ Инструментарий


⌘ Работа с различными типами памяти

⌘ Паттерны программирования на CUDA

⌘ Разное

Процесс разработки программ CUDA

Общие рекомендации по оптимизации



- ⌘ Переосмысление задачи в терминах параллельной обработки данных
 - ☑ Выявляйте параллелизм
 - ☑ Максимизируйте интенсивность вычислений
 - ☒ Иногда выгоднее пересчитать чем сохранить
 - ☑ Избегайте лишних транзакций по памяти
- ⌘ Особое внимание особенностям работы с различными видами памяти (об этом дальше)
- ⌘ Эффективное использование вычислительной мощности
 - ☑ Разбивайте вычисления с целью поддержания сбалансированной загрузки SM'ов
 - ☑ Параллелизм потоков vs. параллелизм по данным

Содержание



⌘ Процесс разработки программ CUDA

- ☐ Портирование части приложения
- ☐ Общие рекомендации по оптимизации
- ☐ Инструментарий

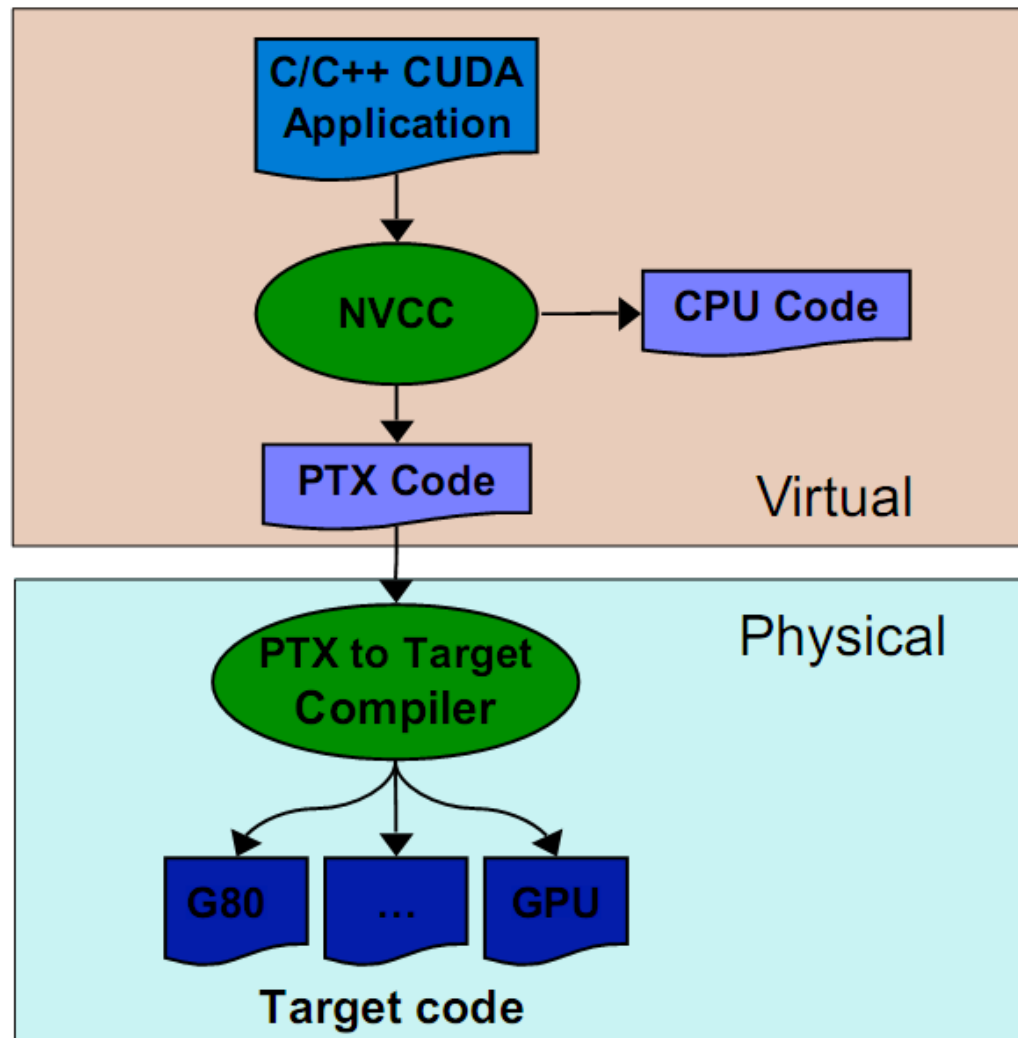
⌘ Работа с различными типами памяти

⌘ Паттерны программирования на CUDA

⌘ Разное

Процесс разработки программ CUDA

Инструментарий: Компилятор



Процесс разработки программ CUDA

Инструментарий: Компилятор

- ⌘ Статическая компиляция: IDE(MS Visual Studio + cuda.rules), Makefile, CL
- ⌘ PTX JIT-компиляция

```
//Device Code, file: myKernel.cu
__global__ void myCUfunction_name(int *dst)
{
    int addr = blockIdx.x * blockDim.x + threadIdx.x;
    dst[addr] = addr >> 1;
}
```

nvcc myKernel.cu -ptx

myKernel.ptx

```
.entry myCUfunction_name(.param .u32 dst)
{
    .reg .u16 %rh<4>;
    .reg .u32 %r<12>;
    .loc 14 2 0
    mov.u16 %rh1, %ctaid.x;
    ...
}
```

From resource

```
const unsigned char
PTXdump[] = {0x00, 0x01};
```

cuModuleLoad

PTX JIT

cuModuleLoadData

Процесс разработки программ CUDA

Инструментарий: Компилятор

Device code

```
__global__ void myCUfunction_name(int *dst)
{
    int addr = blockIdx.x * blockDim.x + threadIdx.x;
    dst[addr] = addr >> 1;
}
```

Host code: driver API + PTX JIT compilation

```
extern unsigned char PTXdump[];
```

```
cuModuleLoadData(&myCUmodule,
    PTXdump);
```

```
cuModuleGetFunction(myCUfunction,
    myCUmodule,
    "myCUfunction_name");
```

```
cuFuncSetBlockShape(myCUfunction, blockDimX);
cuParamSeti(myCUfunction, 0, srcDevPtr);
cuParamSetSize(myCUfunction, 4);
```

```
cuLaunchGrid(myCUfunction, gridDimX, gridDimY);
cuCtxSynchronize();
```

Процесс разработки программ CUDA

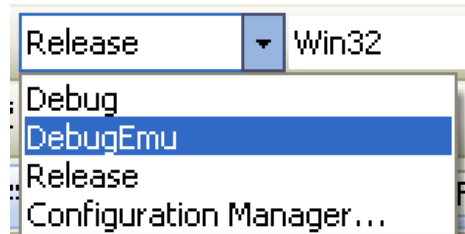
Инструментарий: Отладчик

⌘ GPU debugger

- ☒ Wednesday, April 08: Today NVIDIA announces an industry milestone for GPU Computing. With CUDA 2.2 beta we are including the industries 1st GPU HW Debugger to our developer community.

⌘ GPU emulation

- ☒ `-deviceemu D_DEVICEEMU`
- ☒ Запускает по одному host-процессу на каждый CUDA-поток
- ☒ Работоспособность в режиме эмуляции не всегда коррелирует с работоспособностью на GPU



⌘ Два инструмента не конкурируют, а дополняют друг друга

- ☒ Один из интересных сценариев: Boundchecker + Emulation

Процесс разработки программ CUDA


Инструментарий: Отладчик

⌘ Достоинства эмуляции

- ☑ Исполняемый файл, скомпилированный в режиме эмуляции работает целиком на CPU
 - ☒ Не требуется драйвер CUDA и GPU
 - ☒ Каждый поток GPU эмулируется потоком CPU
- ☑ При работе в режиме эмуляции можно:
 - ☒ Использовать средства отладки CPU (точки останова и т.д.)
 - ☒ Обращаться к любым данным GPU с CPU и наоборот
 - ☒ Делать любые CPU-вызовы из код GPU и наоборот (например `printf()`)
 - ☒ Выявлять ситуации зависания, возникающие из-за неправильного применения `__syncthreads()`

Процесс разработки программ CUDA

Инструментарий: Отладчик



⌘ Недостатки эмуляции

- ☒ Часто работает очень медленно
- ☒ Неумышленное разыменованное указателей GPU на стороне CPU или наоборот
- ☒ Результаты операций с плавающей точкой CPU и «настоящего» GPU почти всегда различаются из-за:
 - ☒ Разного порядка выполняемых операций
 - ☒ Разных допустимых ошибок результатов
 - ☒ Использования большей точности при расчёте промежуточных результатов на CPU

Процесс разработки программ CUDA

Инструментарий: Профилировщик

⌘ CUDA Profiler, позволяет отслеживать:

- ☑ Время исполнения на CPU и GPU в микросекундах
- ☑ Конфигурацию grid и thread block
- ☑ Количество статической разделяемой памяти на блок
- ☑ Количество регистров на блок
- ☑ Коэффициент занятости GPU (Occupancy)
- ☑ Количество объединенных и индивидуальных запросов к глобальной памяти (coalescing)
- ☑ Количество дивергентных путей исполнения (branching)
- ☑ Количество выполненных инструкций
- ☑ Количество запущенных блоков

⌘ Вся эта информация собирается с первого SM или TPC.
Профилирование Uber-kernel'ов с осторожностью

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
 - ☒ Константная
 - ☒ Текстурная
 - ☒ Глобальная
 - ☒ Разделяемая
- ⌘ Паттерны программирования на CUDA
- ⌘ Разное

Работа с константной памятью



- ⌘ Быстрая, кешируемая, только для чтения
- ⌘ Данные должны быть записаны до вызова кернела (например при помощи **cudaMemcpyToSymbol**)
- ⌘ Всего 64Kb (Tesla)
- ⌘ Объявление при помощи слова __constant__
- ⌘ Доступ из device кода простой адресацией
- ⌘ Срабатывает за 4 такта на один адрес внутри варпа
 - ⏏ 4 такта на всю транзакцию в случае если все потоки внутри варпа читают один адрес
 - ⏏ В худшем случае 64 такта

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
 - ☒ Константная
 - ☒ Текстурная
 - ☒ Глобальная
 - ☒ Разделяемая
- ⌘ Паттерны программирования на CUDA
- ⌘ Разное

Работа с текстурной памятью

- ⌘ Быстрая, кешируемая в 2-х измерениях, только для чтения
- ⌘ Данные должны быть записаны при помощи `cudaMemcpyToArray`, либо возможно прикрепление к глобальной памяти через `cudaBindTexture2D`
- ⌘ Объявление при помощи текстурных ссылок
- ⌘ Доступ из device кода при помощи `tex1D`, `tex2D`, `tex1Dfetch`
- ⌘ Лучшая производительность при условии что потоки одного варпа обращаются к локализованной окрестности в 2D

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
 - ☒ Константная
 - ☒ Текстурная
 - ☒ Глобальная
 - ☒ Разделяемая
- ⌘ Паттерны программирования на CUDA
- ⌘ Разное

Работа с глобальной памятью



- ⌘ Медленная, некешируемая (G80), чтение/запись
- ⌘ Запись данных с/на хост через **cudaMemcpy***
- ⌘ Транзакции по PCI-е медленные: макс. 4GB/s vs. 80 GB/s при копировании device-device
- ⌘ Возможность асинхронных транзакций
- ⌘ Ускорение транзакций путем выделения host page-locked памяти (**cudaMallocHost**)
- ⌘ Объявление при помощи слова **__global__**
- ⌘ Доступ простой индексацией
- ⌘ Время доступа от 400 до 600 тактов на транзакцию – высокая латентность

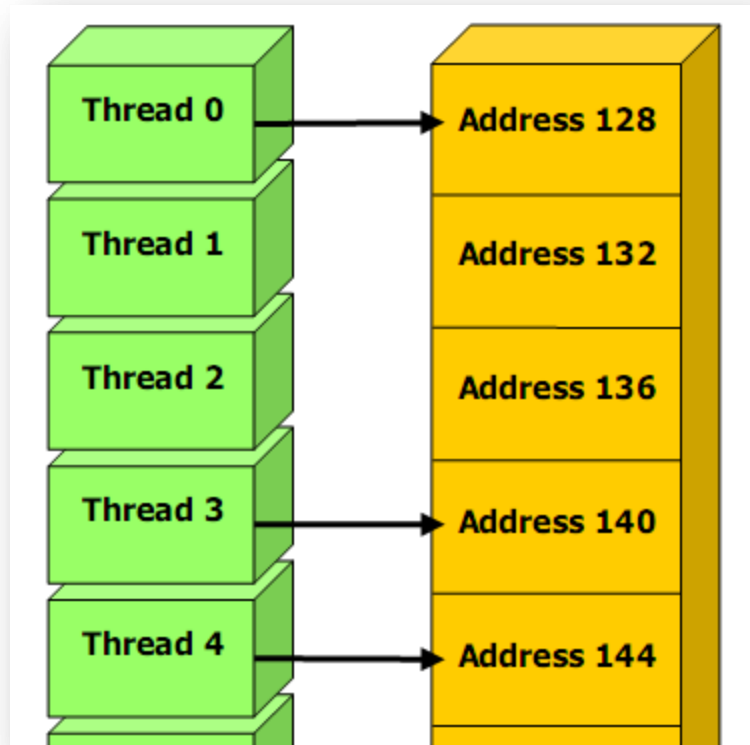
Работа с глобальной памятью

Coalescing, Compute Capability 1.0, 1.1

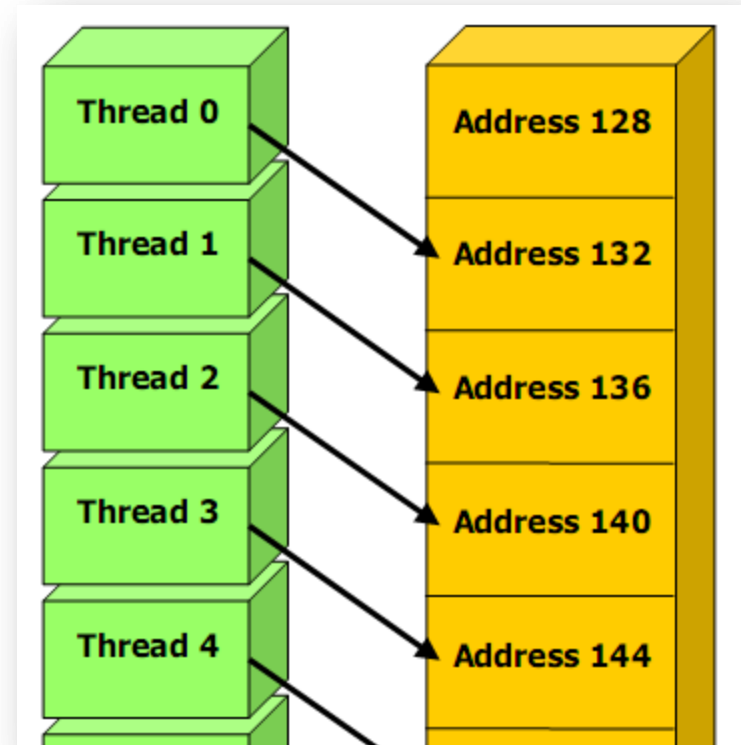
- ⌘ 16 потоков. Типы транзакций:
 - ⏏ 4-байтовые слова, одна 64-байтовая транзакция
 - ⏏ 8-байтовые слова, одна 128-байтовая транзакция
 - ⏏ 16-байтовые слова, две 128-байтовых транзакции
- ⌘ Все 16 элементов должны лежать в едином сегменте, размер и выравнивание которого совпадает с размером транзакции
- ⌘ Строгий порядок доступа: k-й поток обращается к k-му элементу в сегменте
- ⌘ При нарушении порядка вместо одной транзакции получается 16
- ⌘ Некоторые из потоков могут не участвовать

Работа с глобальной памятью

Coalescing, Compute Capability 1.0, 1.1



Coalescing



No coalescing

Работа с глобальной памятью

Coalescing, Compute Capability 1.2, 1.3



⌘ Объединенная транзакция получается, если все элементы лежат в сегментах:

☑ размера 32 байта, потоки обращаются к 1-байтовым элементам

☑ размера 64 байта, потоки обращаются к 2-байтовым элементам

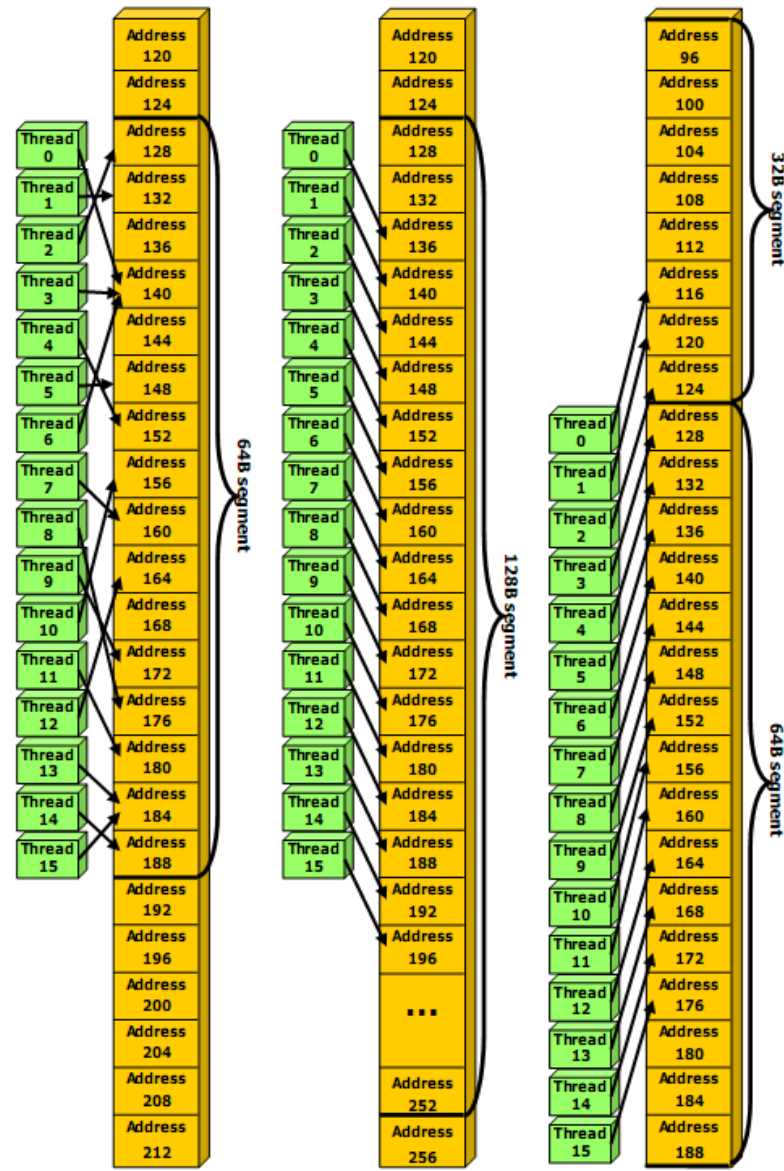
☑ размера 128 байт, потоки обращаются к 4- и 8-байтовым элементам

⌘ Нестрогий порядок доступа. Возможно обращение несколькими потоками к одному адресу

⌘ При выходе за границы сегмента число транзакций увеличивается минимально

Работа с глобальной памятью

Coalescing, Compute Capability 1.2, 1.3



Работа с глобальной памятью

Coalescing. Рекомендации

- ⌘ Используйте **cudaMallocPitch** для работы с 2D-массивами
- ⌘ Конфигурируйте блоки с большей протяженностью по **x**
- ⌘ Параметризуйте конфигурацию, экспериментируйте
- ⌘ В сложных случаях используйте привязку сегмента глобальной памяти к текстуре в случае если Compute Capability < 1.2
 - ⏏ **cudaBindTexture, tex1Dfetch**
 - ⏏ **cudaBindTexture2D, tex2D**

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
 - ☒ Константная
 - ☒ Текстурная
 - ☒ Глобальная
 - ☒ Разделяемая
- ⌘ Паттерны программирования на CUDA
- ⌘ Разное

Работа с разделяемой памятью

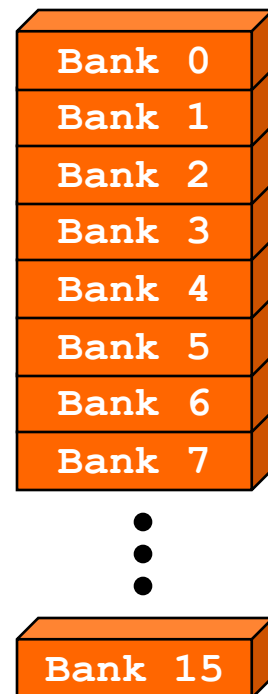


- ⌘ Быстрая, некешируемая, чтение/запись
- ⌘ Объявление при помощи слова shared
- ⌘ Доступ из device кода при помощи индексирования
- ⌘ Самый быстрый тип памяти после регистров, низкая латентность доступа
- ⌘ Можно рассматривать как полностью открытый L1-кеш
- ⌘ При работе с разделяемой памятью следует помнить о ее разбиении на банками памяти

Работа с разделяемой памятью

Банки памяти

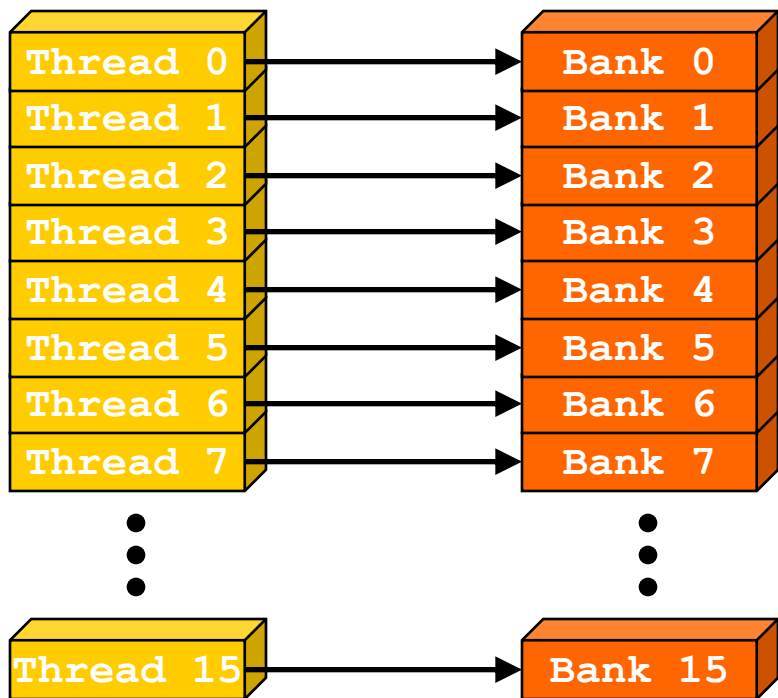
- ⌘ Память разделена на 16 банков памяти, по числу потоков в варпе
- ⌘ Каждый банк может обратиться к одному адресу за 1 такт
- ⌘ Максимальное число адресов, к которым может обратиться память одновременно совпадает с числом банков
- ⌘ Одновременное обращение нескольких потоков из одного полуварпа к одному банку приводит к конфликту банков и сериализации запросов (кроме broadcast)



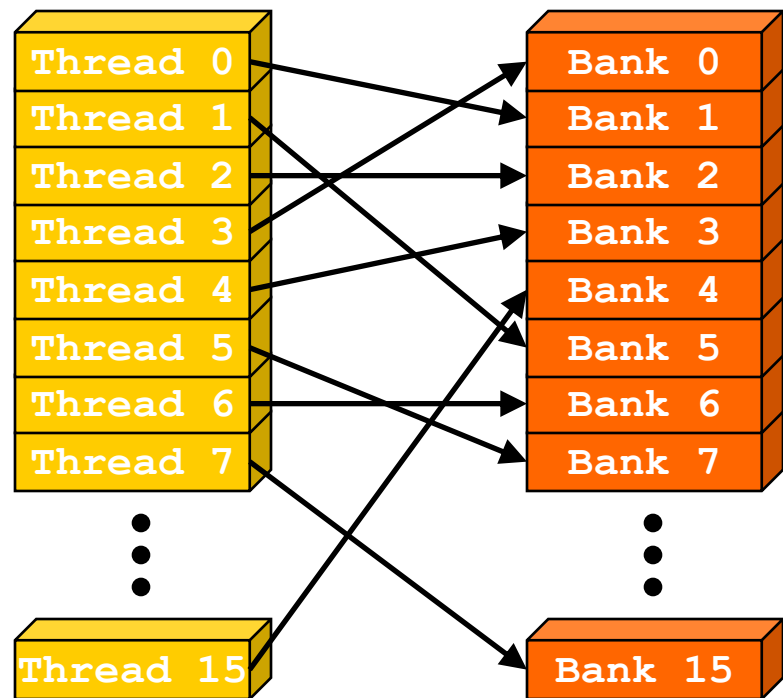
Работа с разделяемой памятью

Банки памяти

⌘ Доступ без конфликтов банков



Прямой доступ

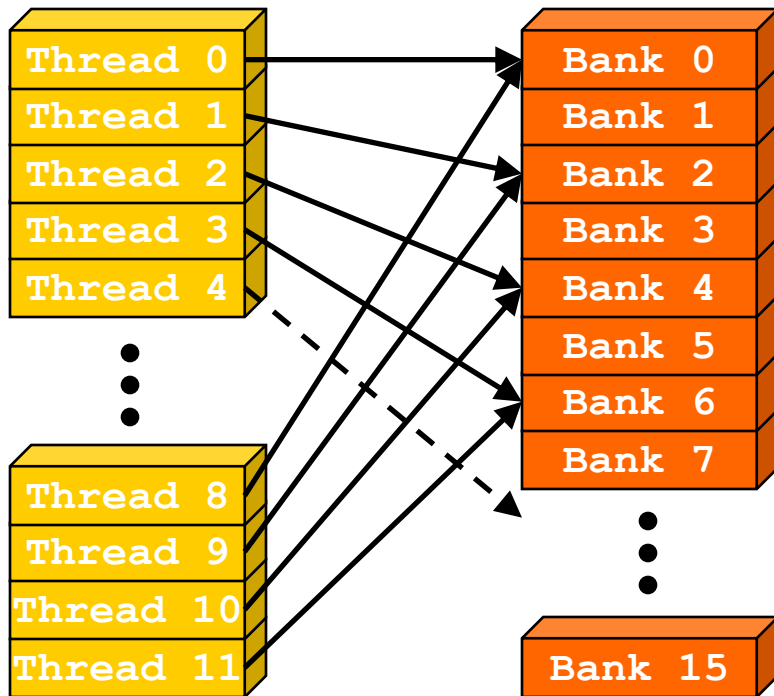


Смешанный доступ 1:1

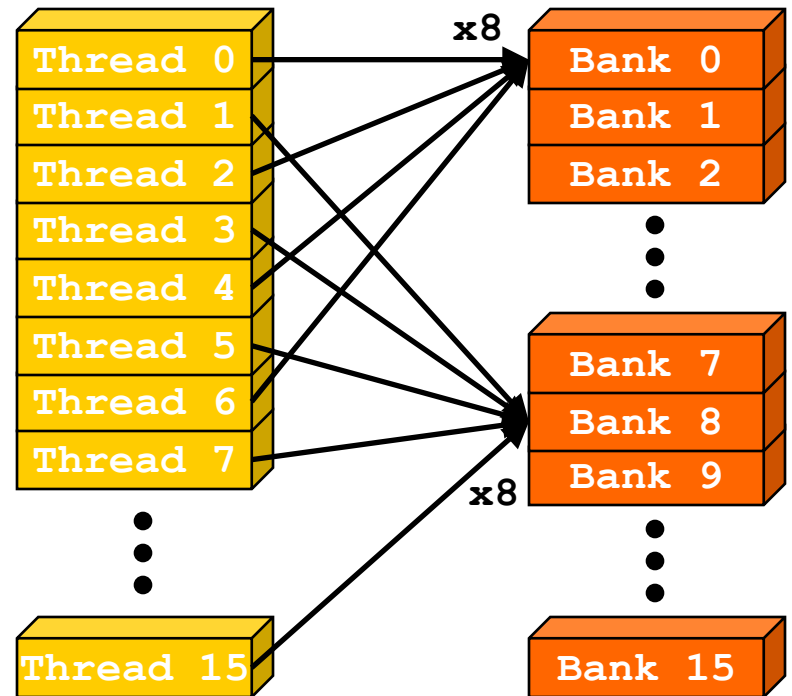
Работа с разделяемой памятью

Банки памяти

⌘ Доступ с конфликтами банков



2-кратный конфликт



8-кратный конфликт

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
- ⌘ Паттерны программирования на CUDA
 - ☒ Приоритеты оптимизации
 - ☒ Сценарий работы с shared памятью
 - ☒ Копирование global <-> shared
 - ☒ Обработка в shared памяти
- ⌘ Разное

Паттерны программирования на CUDA

Приоритеты оптимизации



- ⌘ Объединение запросов к глобальной памяти
 - ☒ Ускорение до 20 раз
 - ☒ Стремление к локальности
- ⌘ Использование разделяемой памяти
 - ☒ Высокая скорость работы
 - ☒ Удобство взаимодействия потоков
- ⌘ Эффективное использование параллелизма
 - ☒ GPU не должен простаивать
 - ☒ Преобладание вычислений над операциями с памятью
 - ☒ Много блоков и потоков в блоке
- ⌘ Банк-конфликты
 - ☒ Если избавление от 4-кратных конфликтов банков влечет увеличение числа инструкций, то данный вид оптимизации можно не делать

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
- ⌘ Паттерны программирования на CUDA
 - ☑ Приоритеты оптимизации
 - ☑ Сценарий работы с shared памятью
 - ☑ Копирование global <-> shared
 - ☑ Обработка в shared памяти
- ⌘ Разное

Паттерны программирования на CUDA

Сценарий работы с shared памятью



1. Загрузка данных из глобальной памяти в разделяемой
 2. `__syncthreads()` ;
 3. Обработка данных в разделяемой памяти
 4. `__syncthreads()` ; //если требуется
 5. Сохранение результатов в глобальной памяти
-
- ⌘ Шаги 2–4 могут быть обрaмлены в условия и циклы
 - ⌘ Шаг 4 может быть ненужен в случае если выходные данные независимы между собой


Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
- ⌘ Паттерны программирования на CUDA
 - ☑ Приоритеты оптимизации
 - ☑ Сценарий работы с shared памятью
 - ☑ Копирование global <-> shared
 - ☑ Обработка в shared памяти
- ⌘ Разное

Паттерны программирования на CUDA

Копирование global <-> shared: 32-bit



```
dim3 block(64);
```

```
__shared__ float dst[64];
```

```
__global__ void kernel(float *data)
```

```
{ //coalescing, no bank conflicts
```

```
    dst[threadIdx.x] = data[threadIdx.x];
```

```
}
```

Паттерны программирования на CUDA

Копирование global <-> shared: 8-bit

```
dim3 block(64);  
__shared__ byte dst[64];
```

```
__global__ void kernel_bad(byte *data)  
{//no coalescing, 4-way bank conflicts present  
    dst[threadIdx.x] = data[threadIdx.x];  
}
```

```
__global__ void kernel_good(byte *data)  
{//coalescing, no bank conflicts, no branching  
    if (threadIdx.x < 16)  
    {  
        int tx = threadIdx.x * 4;  
        *((int *) (dst + tx)) = *((int *) (data + tx));  
    }  
}
```

Содержание



- ⌘ Процесс разработки программ CUDA
- ⌘ Работа с различными типами памяти
- ⌘ Паттерны программирования на CUDA
 - ☑ Приоритеты оптимизации
 - ☑ Сценарий работы с shared памятью
 - ☑ Копирование global <-> shared
 - ☑ Обработка в shared памяти
- ⌘ Разное

Паттерны программирования на CUDA

Обработка в shared памяти

```
__shared__ byte buf[64];  
dim3 block(64);
```

Независимая обработка элементов. Прямой доступ будет вызывать 4-кратный конфликт банков.

Задача: переформировать потоки в 4 группы по 16 индексов так, чтобы при новой косвенной адресации не было конфликтов банков.

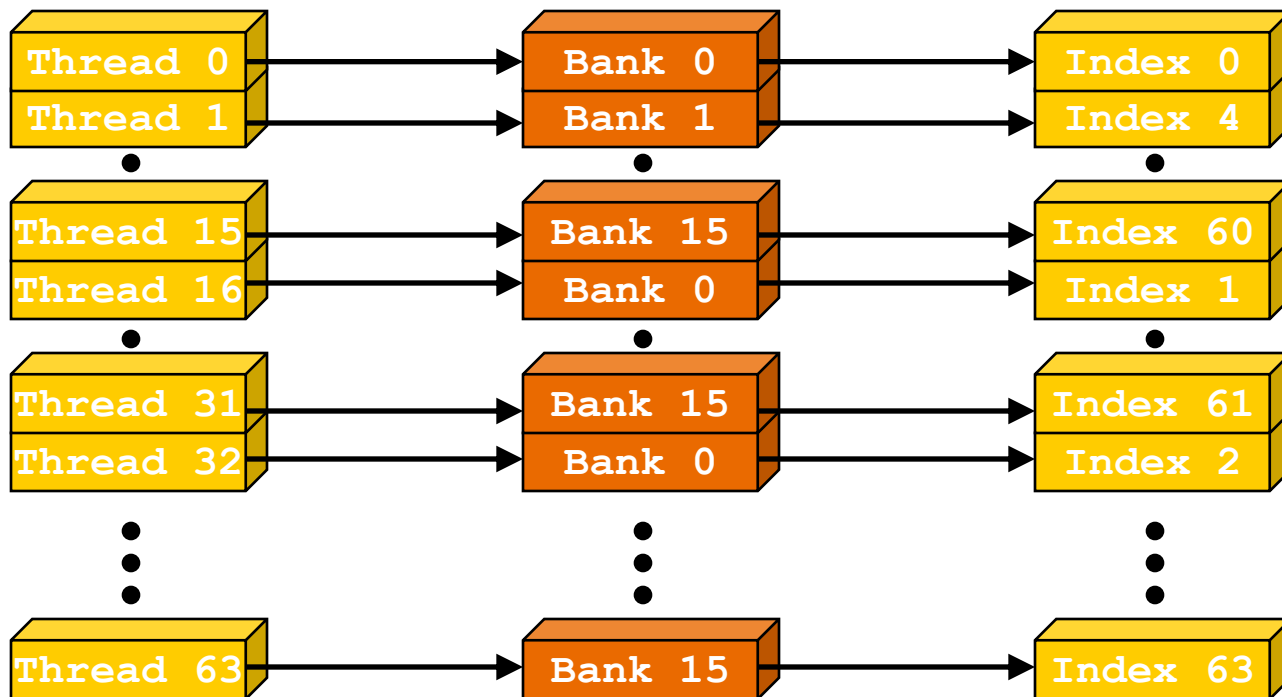


Паттерны программирования на CUDA

Обработка в shared памяти

Одно из решений:

```
__device__ int permute64by4(int t)
{
    return (t >> 4) + ((t & 0xF) << 2);
}
```



Паттерны программирования на CUDA

Обработка в shared памяти (2)



```
__shared__ int buf[16][16];  
dim3 block(16,16);
```

Независимая обработка элементов. Прямой доступ будет вызывать 16-кратный конфликт банков.

Задача: свести число банк-конфликтов до нуля.

Паттерны программирования на CUDA

Обработка в shared памяти (2)

Одно из решений:

```
__shared__ int buf[16][17];  
dim3 block(16,16);
```

Bank Indices without Padding

0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	2	3	4	5	6	7	...	15

Bank Indices with Padding

0	1	2	3	4	5	6	7	...	15	0
1	2	3	4	5	6	7	8	...	0	1
2	3	4	5	6	7	8	9	...	1	2
3	4	5	6	7	8	9	10	...	2	3
4	5	6	7	8	9	10	11	...	3	4
5	6	7	8	9	10	11	12	...	4	5
6	7	8	9	10	11	12	13	...	5	6
7	8	9	10	11	12	13	14	...	6	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	0	1	2	3	4	5	6	...	14	15

Разное



⌘ Конфигурация gridDim и blockDim возможно во время исполнения:

```
void callKernel(dim3 grid, dim3 threads)
{
    kernel<<<grid, threads>>>();
}
```

Разное



⌘ Конфигурация gridDim и blockDim возможно во время исполнения:

```
void callKernel(dim3 grid, dim3 threads)
{
    kernel<<<grid, threads>>>();
}
```

Разное



- ⌘ `__mul24` и `__umul24` работают быстрее, чем `*`
- ⌘ Возможно увеличение числа регистров после применения
- ⌘ На будущих архитектурах ситуация может развернуться наоборот и `__mul24` станет медленнее
- ⌘ В остальном целочисленная арифметика работает примерно с такой же скоростью, как и с плавающей точкой (за исключением целочисленного деления)

Разное



⌘ Математика FPU (на GPU в частности) не ассоциативна

⌘ $(x+y)+z$ не всегда равно $x+(y+z)$

⌘ Например при $x = 10^{30}$, $y = -10^{30}$, $z = 1$

Ресурсы нашего курса



⌘ CUDA.CS.MSU.SU

- ☒ Место для вопросов и дискуссий
- ☒ Место для материалов нашего курса
- ☒ Место для ваших статей!
 - ☒ Если вы нашли какой-то интересный подход!
 - ☒ Или исследовали производительность разных подходов и знаете, какой из них самый быстрый!
 - ☒ Или знаете способы сделать работу с CUDA проще!

⌘ www.steps3d.narod.ru

⌘ www.nvidia.ru

Вопросы

