




Казанский
федеральный
университет

ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

OpenACC Библиотеки CUDA

OpenACC

- ▶ OpenACC – набор инструкций компилятора для параллелизации кода
 - ▶ Позволяет быстро ускорить код без освоения программирования на CUDA
 - ▶ Для работы необходим NVIDIA OpenACC Toolkit
 - ▶ Для работы необходимо оформить лицензию
 - ▶ GCC 7.0 частично поддерживает OpenACC
- 

OpenACC

Приложения

ЯП CUDA


OpenACC

Библиотеки
CUDA

OpenACC

- ▶ OpenACC – не программирование GPU, а выражение параллелизма участков кода
- ▶ Аналог OpenMP для GPU
- ▶ Как правило параллелизуются циклы
- ▶ Пример: SAXPY на OpenACC

Библиотеки CUDA

- ▶ В настоящий момент существует множество специализированных библиотек CUDA
 - ▶ Основной принцип – не изобретать велосипед
 - ▶ Если вам нужно решить задачу – скорее всего ее уже решили до вас
 - ▶ Как правило библиотечная реализация отлажена, оптимизирована и протестирована
- 

Библиотеки CUDA

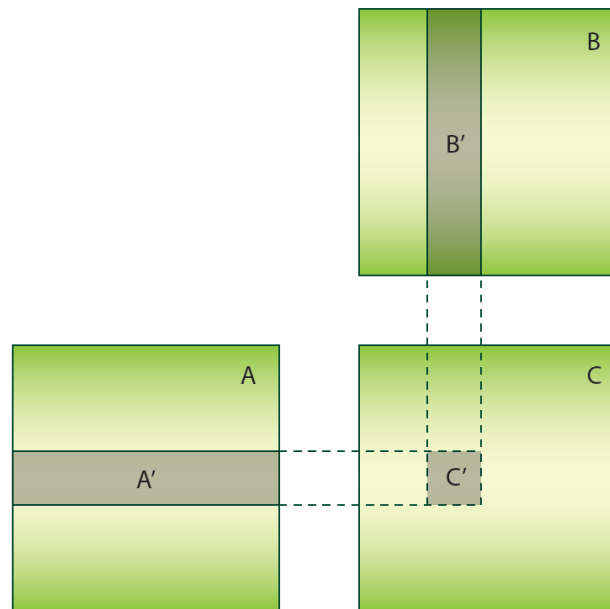
- ▶ Некоторые библиотеки Nvidia
 - CUBLAS – основные операции линейной алгебры
 - CUSPARSE – основные операции линейной алгебры для разреженных матриц
 - CUFFT – прямое и обратное быстрое ДПФ
 - CURAND – генерация псевдослучайных чисел
 - Thrust – аналог STL для CUDA
 - OpenACC – аналог OpenMP для CUDA

Перемножение матриц

- ▶ Квадратные матрицы $n \times n$
- ▶ Допустим n кратно 32
- ▶ Наивный алгоритм:
 - Аналог последовательного алгоритма
 - 2d блоки (32×32) и 2d сетка блоков
 - Каждая нить вычисляет свой элемент матрицы
 - На каждый элемент $2n$ обращений к глобальной памяти

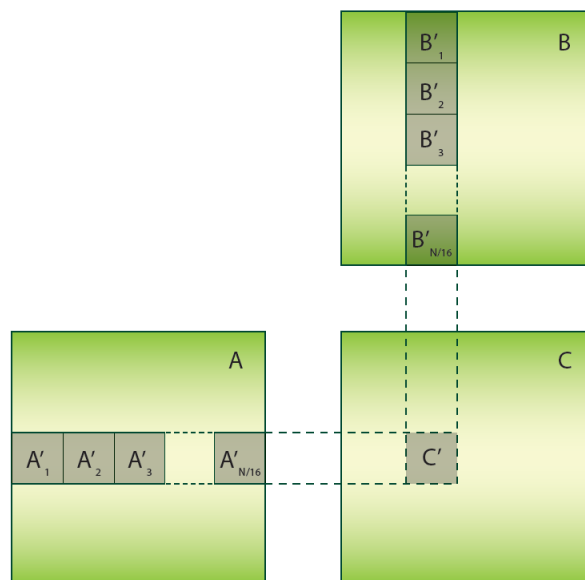
Перемножение матриц

- ▶ Разобьем вычисления на блоки
- ▶ Для вычисления подматрицы C' (32×32) необходимы полосы A' ($n \times 32$) и B' ($32 \times n$)



Перемножение матриц

- ▶ Размер разделяемой памяти ограничен – полосы могут не поместиться при больших n
- ▶ Разобьем полосы на подматрицы 32×32
- ▶ C' – сумма произведений подматриц



Перемножение матриц


- ▶ Вариант с использованием функции `cublasSgemm`

- ▶ Выполняет операцию

$$C = \alpha \cdot A \times B + \beta \cdot C$$

- ▶ Матрицы A и B могут быть транспонированы
- ▶ Матрицы хранятся по столбцам
- ▶ Компиляция с ключом `-lcublas`

Thrust

- ▶ Библиотека шаблонов для CUDA C++ основанная на STL
 - ▶ Позволяет быстро создавать параллельный код через высокоуровневый интерфейс
 - ▶ Совместим с CUDA C
 - ▶ Позволяет писать код в ОО и функциональном стиле
- 

Thrust

- ▶ Предоставляет большую коллекцию параллельных алгоритмов
 - Сканирование (scan)
 - Свертка (reduce)
 - Сортировка
 - etc.
- ▶ Контейнеры
 - `host_vector`
 - `device_vector`

Контейнеры thrust

```
int main(void)
{
    //инициализируем вектор на GPU единицами
    thrust::device_vector<int> D(10, 1);
    //установим первые 7 элементов равными 9
    thrust::fill(D.begin(), D.begin() + 7, 9);
    //создадим вектор на хосте на основе 5 элементов D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);
    // установим элементы H равными 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());
    //скопируем все элементы H в начало D
    thrust::copy(H.begin(), H.end(), D.begin());
    //распечатаем D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;
    return 0;
}
```

Контейнеры thrust

```
int main(void)
{
    //создаем list из STL
    std::list<int> stl_list;
    stl_list.push_back(10);
    stl_list.push_back(20);
    stl_list.push_back(30);
    stl_list.push_back(40);

    //инициализируем device_vector с помощью list
    thrust::device_vector<int> D(stl_list.begin(), stl_list.end());

    //копируем device_vector в STL-вектор
    std::vector<int> stl_vector(D.size());
    thrust::copy(D.begin(), D.end(), stl_vector.begin());

    return 0;
}
```

Трансформации в thrust

► Трансформация

- Берет все элементы некоего входного множества
- Применяет к ним определенную операцию – функтор
- Записывает результат по указанному адресу

```
thrust::device_vector<int> X(10);  
thrust::device_vector<int> Y(10);  
  
thrust::sequence(X.begin(), X.end());  
  
//Y = -X  
thrust::transform(X.begin(), X.end(), Y.begin(),  
thrust::negate<int>());
```

Трансформации в thrust

- ▶ Функторы покрывают большинство встроенных арифметических операций
- ▶ Можно создавать пользовательские функторы
- ▶ Пример – использование трансформации для операции SAXPY
 - Медленный вариант: использование 2-х трансформаций и дополнительного вектора
 - Быстрый: использование собственного функтора

Редукция в thrust

- ▶ В thrust реализована операция редукции
- ▶ Пример: редукция по сумме из Лекции 3
- ▶ Кроме стандартных сверток, в thrust есть
 - `thrust::inner_product`
 - `thrust::max_element` / `thrust::min_element`
 - `thrust::is_sorted`
 - `thrust::inner_product`
 - etc. (RTFM)

Редукция в thrust

- `thrust::transform_reduce` – применение трансформации к последовательности, затем применение свертки к результату

```
//компиляция с ключом --expt-extended-lambda
auto un_op = []__device__(float &x){return sin(x)};

thrust::multiplies<float> bin_op;

thrust::transform_reduce(
    d_x.begin(),
    d_x.end(),
    un_op,
    1,
    bin_op
);
```

Сортировка в thrust

- ▶ Thrust позволяет легко сортировать массивы на GPU
- ▶ Пример: сортировка массива float длиной 2^{27}
- ▶ Обратите внимание: затраты на копирование данных значительно больше чем на сортировку

Итераторы thrust

- ▶ В thrust предусмотрена взаимная совместимость данных с CUDA API
 - Преобразование итератора Thrust в указатель CUDA C
 - Преобразование указателя CUDA C в итератор Thrust

Итераторы thrust

```
size_t n = 1 << 20;
int * raw_ptr;
//выделяем память по указателю
cudaMalloc(&raw_ptr, n * sizeof(int));

//создаем итератор на основе указателя
thrust::device_ptr<int> dev_ptr(raw_ptr);

//используем итератор
thrust::fill(dev_ptr, dev_ptr + n, (int) 0);
```

```
size_t n = 1 << 20;
//создаем вектор на устройстве
thrust::device_vector<int> d_vec(n);

//получаем указатель на память вектора
int* ptr = thrust::raw_pointer_cast(&d_vec[0]);

//используем указатель при вызове ядра
my_kernel<<< n / 256, 256 >>>(n, ptr);
```

Итераторы thrust

- ▶ В thrust существуют fancy iterators – специальные итераторы в стиле Boost.Iterator
 - `constant_iterator` – всегда возвращает одно и то же значение
 - `transform_iterator` – позволяет объединять несколько преобразований (типа `transform_reduce`)
 - `zip_iterator` – позволяет объединять последовательности в кортежи (`tuple`) и передавать их в алгоритмы thrust



Казанский
федеральный
университет

ВЫСШАЯ ШКОЛА
информационных технологий
и информационных систем

Вопросы

ekhramch@kpfu.ru