

БИБЛИОТЕКИ С ПОДДЕРЖКОЙ CUDA

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

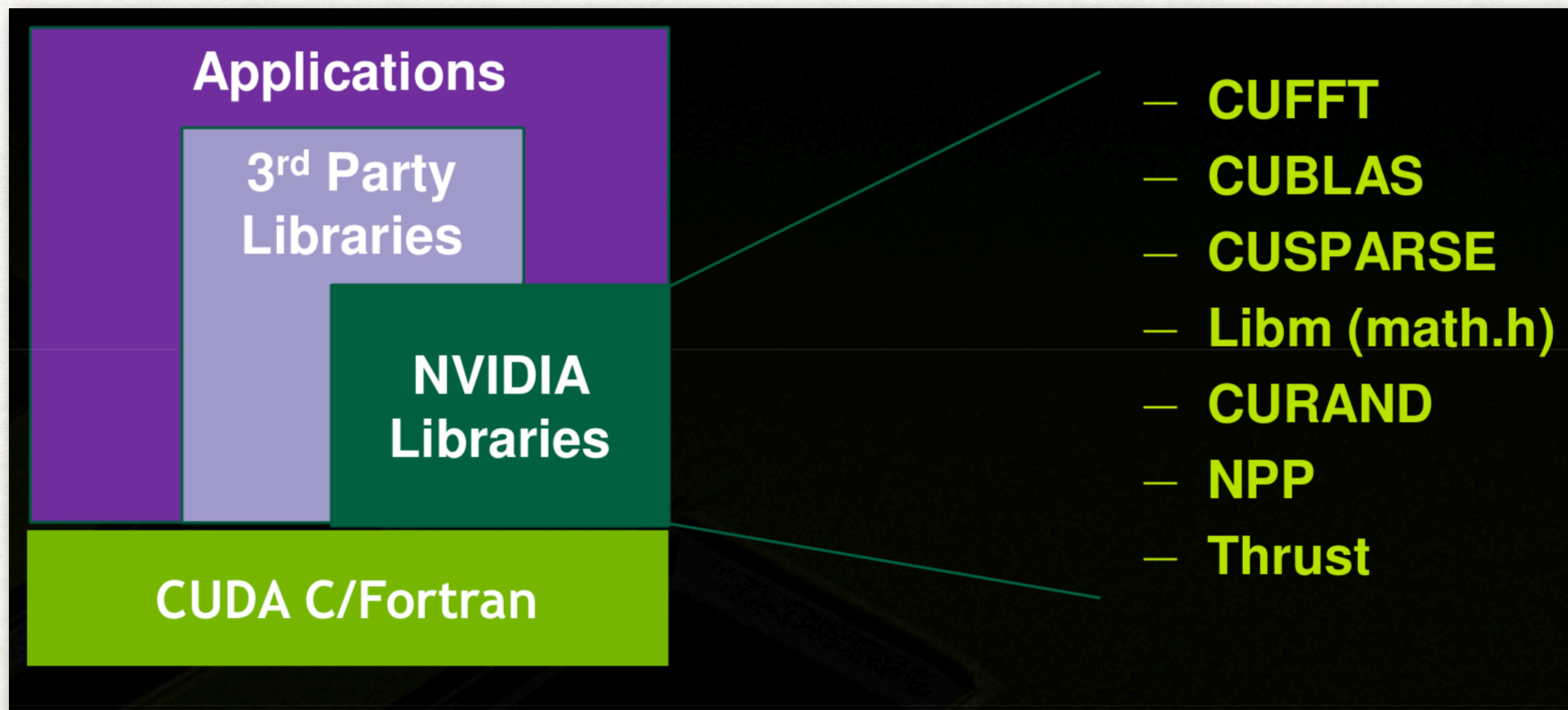
CUDA LIBRARIES

- Представляют собой готовые к использованию базовые блоки для построения программ
- Просты в использовании
- Оптимизированы для максимальной производительности
- Идеальны для приложений, которые уже используют аналогичные CPU библиотеки

CUDA LIBRARIES

- Доступны следующие библиотеки:
 - cuFFT - Fast Fourier Transforms Library
 - cuBLAS - Complete BLAS Library
 - cuSPARSE - Sparse Matrix Library
 - cuRAND - Random Number Generation (RNG) Library
 - NPP - Performance Primitives for Image & Video Processing
 - Thrust - Templated C++ Parallel Algorithms & Data
 - Structures math.h - C99 floating-point Library
 - NVGraph - graph processing library
- Включены в CUDA Toolkit

CUDA LIBRARIES



О ЧЁМ СЕГОДНЯ ПОЙДЕТ РЕЧЬ

- Краткий обзор функциональности библиотек, примеры их использования
- Для более подробных деталей использования каждой библиотеки следует обратиться к официальной документации

CUBLAS

CUBLAS

- Реализация BLAS (Basic Linear Algebra Subprograms) [?] для GPU

- Поддерживает все функции из BLAS

- Level1 (вектор-вектор): $O(N)$ [?]

- AXPY : $y = \alpha \cdot x + y$

- DOT : $\text{dot} = x \cdot y$

- Level 2(матрица-вектор): $O(N^2)$

- Vector multiplication by a General Matrix : GEMV

- Triangular solver : TRSV

- Level3(матрица-матрица): $O(N^3)$

- General Matrix Multiplication : GEMM

- Triangular Solver : TRSM

- Как и BLAS, CUBLAS хранит матрицы по столбцам, а не строкам (как в Fortran)

CUBLAS

- Поддерживает 4 типа – Float, Double, Complex, Double Complex – соответствующие префиксы: S, D, C, Z
- Содержит 152 процедуры : S(37), D(37), C(41), Z(41)
Принцип построения названия функций: cublas + BLAS name
- Пример: cublasSGEMM
 - S: single precision (float)
 - GE: general
 - M: multiplication
 - M: matrix

CUBLAS

- Интерфейс CUBLAS библиотеки расположен в файле `cublas.h`
- Функции имеют имена в соответствии со следующей конвенцией: `cublas + BLAS name` (напр, `cublasSGEMM`)
- Обработка ошибок
- Вычислительные функции CUBLAS не возвращают ошибки
- CUBLAS позволяет получить последнюю произошедшую ошибку при помощи случайных функций
- Вспомогательные функции CUBLAS:
Memory allocation, data transfer
- Пример использования библиотеки будет рассмотрен далее после CURAND

CURAND

CURAND

- Генерировать качественные последовательности случайных чисел параллельно не так просто
- Не делайте это сами, используйте библиотеку!
- Большой набор генераторов случайных чисел для различных распределений: XORWOW, MRG323ka, MTGP32, (scrambled) Sobol, uniform, normal, log-normal
- Двойная и одинарная точность:
- У CURAND есть 2 API:
 - API для CPU: подходит для генерации больших порций случайных чисел на
- API для GPU: в случае, когда случайные числа должны быть генерированы внутри ядра

ИСПОЛЬЗОВАНИЕ CURAND

1. Инициализировать генератор:

```
curandCreateGenerator()
```

2. Устанавливаем seed:

```
curandSetPseudoRandomGeneratorSeed()
```

3. Генерируем данные задав распределение:

```
curandGenerateUniform()/curandGenerateUniformDouble(): Uniform  
curandGenerateNormal()/cuRandGenerateNormalDouble(): Gaussian  
curandGenerateLogNormal/curandGenerateLogNormalDouble(): Log-Normal
```

4. Уничтожаем генератор:

```
curandDestroyGenerator()
```


CURAND API

- Генерируем множество случайных чисел разом на центральном процессоре, CPU API:

```
#include <curand.h>
```

```
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT); curandGenerateUniform(gen, d_data, n);
```

- Генерируем по одному случайному числу на каждой из GPU нитей, GPU API:

```
#include <curand_kernel.h>
```

```
__global__ void generate_kernel(curandState *state)
```

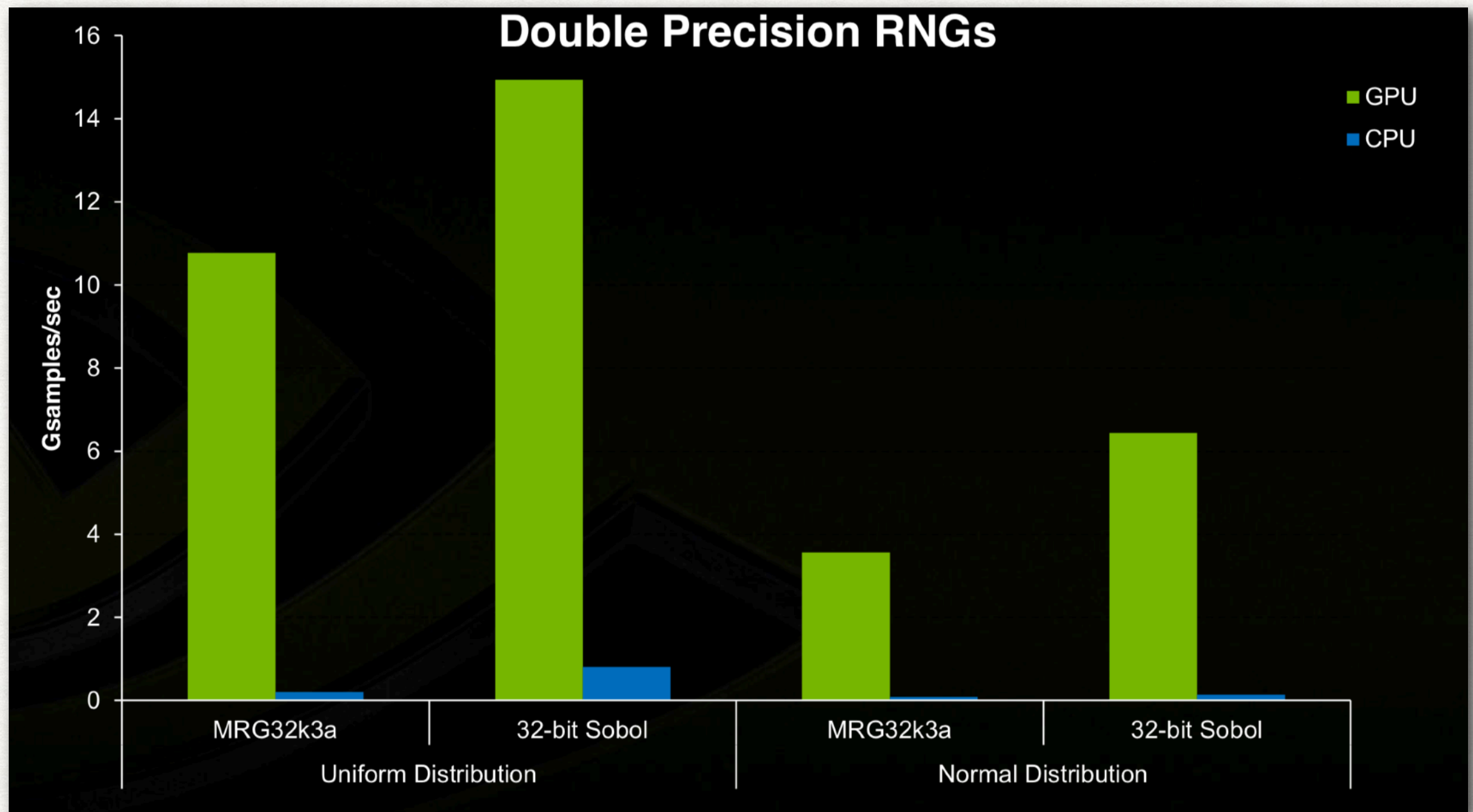
```
{
```

```
    int id = threadIdx.x + blockIdx.x * 64;
```

```
    x = curand(&state[id]);
```

```
}
```


CURAND PERFORMANCE



CUBLAS + CURAND MATRIX MULTIPLICATION EXAMPLE

```
1 #include <cstdlib>
2
3 int main() {
4     // Allocate 3 arrays on CPU
5     int nr_rows_A, nr_cols_A, nr_rows_B, nr_cols_B, nr_rows_C, nr_cols_C;
6
7     // for simplicity we are going to use square arrays
8     nr_rows_A = nr_cols_A = nr_rows_B = nr_cols_B = nr_rows_C = nr_cols_C = 3;
9
10    float *h_A = (float *)malloc(nr_rows_A * nr_cols_A * sizeof(float));
11    float *h_B = (float *)malloc(nr_rows_B * nr_cols_B * sizeof(float));
12    float *h_C = (float *)malloc(nr_rows_C * nr_cols_C * sizeof(float));
13
14    // Allocate 3 arrays on GPU
15    float *d_A, *d_B, *d_C;
16    cudaMalloc(&d_A, nr_rows_A * nr_cols_A * sizeof(float));
17    cudaMalloc(&d_B, nr_rows_B * nr_cols_B * sizeof(float));
18    cudaMalloc(&d_C, nr_rows_C * nr_cols_C * sizeof(float));
19
20    // ....
21
22    //Free GPU memory
23    cudaFree(d_A);
24    cudaFree(d_B);
25    cudaFree(d_C);
26
27    // Free CPU memory
28    free(h_A);
29    free(h_B);
30    free(h_C);
31
32    return 0;
33 }
```


GENERATING RANDOM MATRICES

```
1 ...
2 #include <curand.h>
3 ...
4
5 // Fill the array A(nr_rows_A, nr_cols_A) with random numbers on GPU
6 void GPU_fill_rand(float *A, int nr_rows_A, int nr_cols_A) {
7     // Create a pseudo-random number generator
8     curandGenerator_t prng;
9     curandCreateGenerator(&prng, CURAND_RNG_PSEUDO_DEFAULT);
10
11     // Set the seed for the random number generator using the system
12     curandSetPseudoRandomGeneratorSeed(prng, (unsigned long long)
13     clock());
14
15     // Fill the array with random numbers on the device
16     curandGenerateUniform(prng, A, nr_rows_A * nr_cols_A);
17 }
18 ...
```


PERFORMING MULTIPLICATION

```
1 ...
2 #include <cublas_v2.h>
3 ...
4
5 // Multiply the arrays A and B on GPU and save the result in C
6 //  $C(m,n) = A(m,k) * B(k,n)$ 
7 void gpu_blas_mmul(const float *A, const float *B, float *C, const int m, const int k,
const int n) {
8     int lda=m,ldb=k,ldc=m;
9     const float alf = 1;
10    const float bet = 0;
11    const float *alpha = &alf;
12    const float *beta = &bet;
13
14    // Create a handle for CUBLAS
15    cublasHandle_t handle;
16    cublasCreate(&handle);
17
18    // Do the actual multiplication
19    cublasSgemv(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, alpha, A, lda, B, ldb, beta,
C, ldc);
20
21    // Destroy the handle
22    cublasDestroy(handle);
23 }
```


ИСПОЛЬЗУЕМ THRUST ДЛЯ УПРОЩЕНИЯ ПРИМЕРА С УМНОЖЕНИЕМ МАТРИЦ

```
int main() {
2   // Allocate 3 arrays on CPU
3   int nr_rows_A, nr_cols_A, nr_rows_B, nr_cols_B, nr_rows_C, nr_cols_C;
4
5   // for simplicity we are going to use square arrays
6   nr_rows_A = nr_cols_A = nr_rows_B = nr_cols_B = nr_rows_C = nr_cols_C = 3;
7
8   thrust::device_vector<float> d_A(nr_rows_A * nr_cols_A), d_B(nr_rows_B * nr_cols_B), d_C(nr_rows_C *
nr_cols_C);
9
10  // Fill the arrays A and B on GPU with random numbers
11  GPU_fill_rand(thrust::raw_pointer_cast(&d_A[0]), nr_rows_A, nr_cols_A);
12  GPU_fill_rand(thrust::raw_pointer_cast(&d_B[0]), nr_rows_B, nr_cols_B);
13
14  // Optionally we can print the data
15  std::cout << "A =" << std::endl;
16  print_matrix(d_A, nr_rows_A, nr_cols_A);
17  std::cout << "B =" << std::endl;
18  print_matrix(d_B, nr_rows_B, nr_cols_B);
19
20  // Multiply A and B on GPU
21  gpu_blas_mmul(thrust::raw_pointer_cast(&d_A[0]), thrust::raw_pointer_cast(&d_B[0]),
thrust::raw_pointer_cast(&d_C[0]), nr_rows_A, nr_cols_A, nr_cols_B);
22
23  //Print the result
24  std::cout << "C =" << std::endl;
25  print_matrix(d_C, nr_rows_C, nr_cols_C);
26
27  return 0;
28 }
```


CUSPARSE

CUSPARSE

- Различные форматы хранения разреженных матриц

- ✓ COO (наиболее общий)
- ✓ CSR
- ✓ CSC

- ✓ DIAG

- ✓ ELL
- ✓ HYB (ELL + CSR)

- В основном матрично-векторно операции

- Наименования функций по следующей конвенции:

`cusparse<Type>[<sparse data format>]<operation>[<sparse data format>]`

- Например: single precision, sparse matrix (in csr storage) x dense vector =>
`cusparseScsrmv`

CUSPARSE ФОРМАТЫ

➤ Coordinate (COO) Format

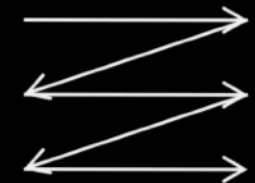
Row Index	1	2	2	3	4	4	4
Col Index	1	1	2	3	1	3	4
Values	1.0	2.0	3.0	4.0	5.0	6.0	7.0

	1	2	3	4
1	1.0			
2	2.0	3.0		
3			4.0	
4	5.0		6.0	7.0

➤ Compressed Sparse Row (CSR)

Row Ptr	1	2	4	5	8		
Col Index	1	1	2	3	1	3	4
Values	1.0	2.0	3.0	4.0	5.0	6.0	7.0

row-major order



➤ Compressed Sparse Column (CSC)

Row Index	1	2	4	2	3	4	4
Col Ptr	1	4	5	7	8		
Values	1.0	2.0	3.0	4.0	5.0	6.0	7.0

column-major order



CUSPARSE EXAMPLES

➤ Level 2 and 3

✓ csrcmv (matrix-vector multiplication)

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \backslash\alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \backslash\beta \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

✓ csrmm (matrix-tall-matrix multiplication)

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} = \backslash\alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 & 5.0 \\ 2.0 & 6.0 \\ 3.0 & 7.0 \\ 4.0 & 8.0 \end{bmatrix} + \backslash\beta \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix}$$

NPP

NPP

- C/c++ примитивы (primitives) – low level API:
- Простая интеграция в код
- Примерно 350 функций обработки изображений
- Примерно 100 функций обработки сигналов

THRUST

THRUST

- Библиотека шаблонов для CUDA – аналогична C++ STL
- Контейнеры – управление памятью на хосте и устройстве:
`thrust::host_vector<T>`, `thrust::device_vector<T>` – позволяет избегать множества типичных ошибок
- Итераторы
 - Автоматически перемещает данные между хостом и устройством
 - `d_vec.begin()`
- Алгоритмы
 - Сортировка, редукция, `scan`, и.т.д.: `thrust::sort()`

THRUST: АЛГОРИТМЫ

- Операции над элементами векторов:
 - `for_each`, `transform`, `gather`, `scatter` ...
- Редукции:
 - `reduce`, `inner_product`, `reduce_by_key` ...
- Префикс-суммы:
 - `inclusive_scan`, `inclusive_scan_by_key` ...
- Сортировки
 - `sort`, `stable_sort`, `sort_by_key` ...

THRUST: ПРИМЕР

```
#include <thrust/host_vector.h>

#include <thrust/device_vector.h>

#include <thrust/sort.h>

#include <cstdlib.h>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);

    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

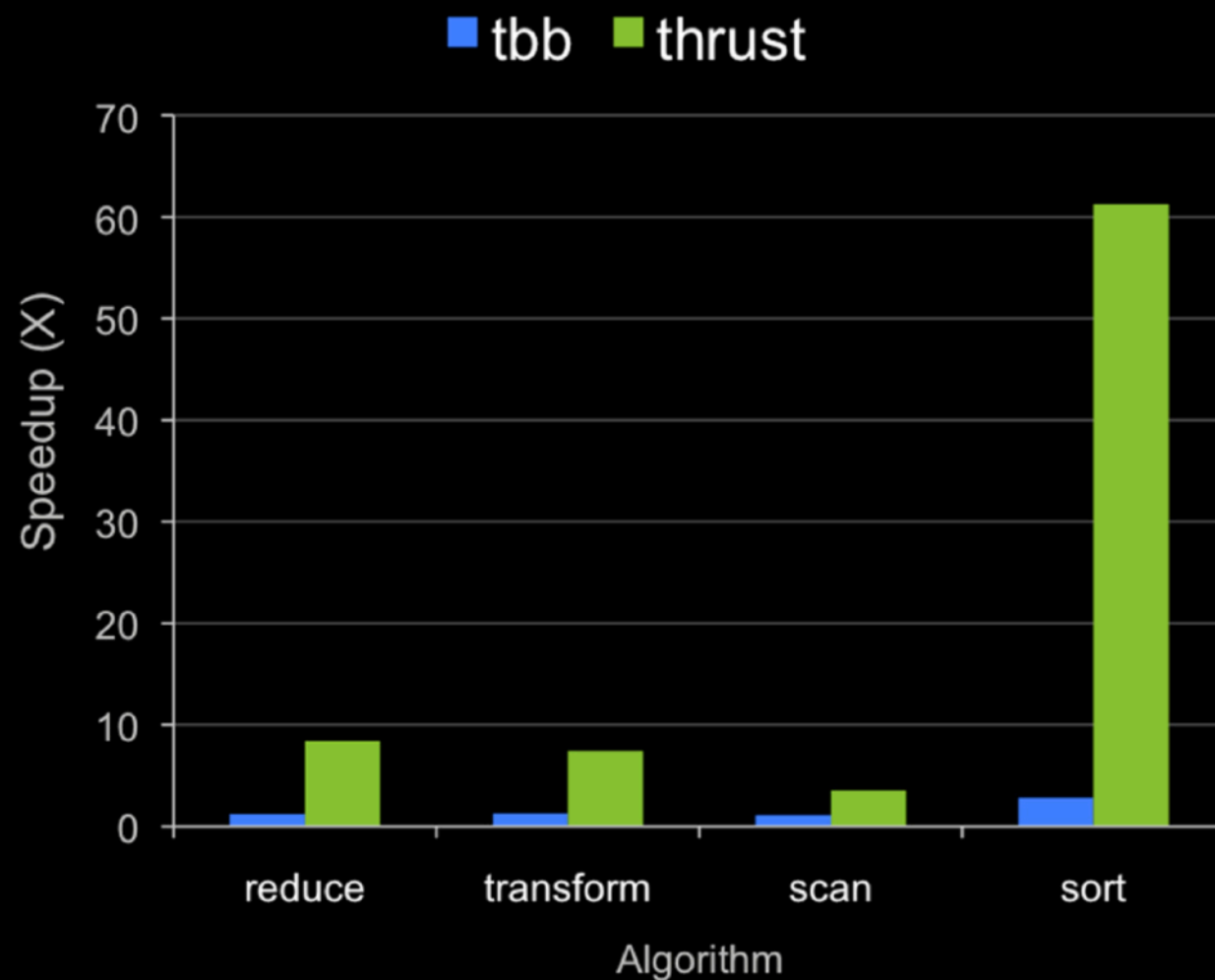
    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

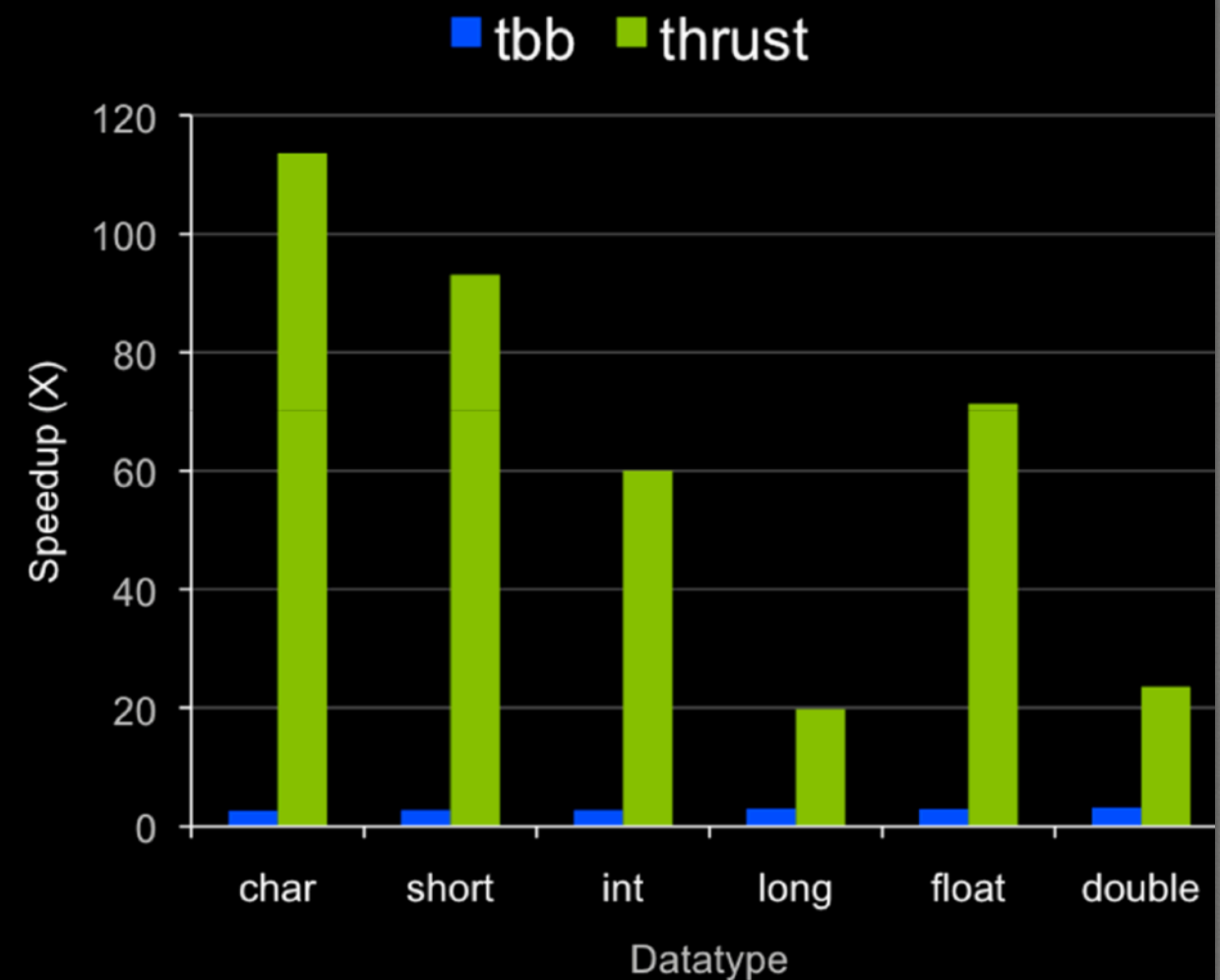
    return 0;
}
```


THRUST: PERFORMANCE

Various Algorithms (32M integers)
Speedup compared to std



Sort (32M samples)
Speedup compared to std



* Thrust 4.0, NVIDIA Tesla C2050 (Fermi)

* Core i7 950 @ 3.07GHz

NVGRAPH

NVGRAPH

- Библиотека аналитики больших графов
- Поддерживает 3 широко-используемых графовых алгоритмов
- **Page Rank**
- **Single Source**
- **Single Source Widest Path**

ГДЕ ИСКАТЬ ИНФОРМАЦИЮ ПО БИБЛИОТЕКАМ?

- В документации CUDA Toolkit - <https://docs.nvidia.com/cuda/index.html>

The screenshot shows the NVIDIA Developer Zone website for the CUDA Toolkit Documentation. The header includes the NVIDIA logo, 'DEVELOPER ZONE', and 'CUDA TOOLKIT DOCUMENTATION'. A search bar is located in the top right corner. The left sidebar contains a navigation menu with categories like 'Turing Tuning Guide', 'CUDA API References', and 'Miscellaneous'. The main content area displays the title 'CUDA Toolkit Documentation v10.0.130' and lists several sections: 'Release Notes', 'EULA', 'Installation Guides' (including Quick Start Guide, Windows, Mac OS X, and Linux), and 'Programming Guides'. A feedback link is also present in the top right of the main content area.

DEVELOPER ZONE **CUDA TOOLKIT DOCUMENTATION**

CUDA Toolkit Documentation - v10.0.130 ([older](#)) - Last updated September 19, 2018 - [Send Feedback](#)

CUDA Toolkit Documentation v10.0.130

Release Notes
The Release Notes for the CUDA Toolkit.

EULA
The End User License Agreements for the NVIDIA CUDA Toolkit, the NVIDIA CUDA Samples, the NVIDIA Display Driver, and NVIDIA NSight (Visual Studio Edition).

Installation Guides

Quick Start Guide
This guide provides the minimal first-steps instructions for installation and verifying CUDA on a standard system.

Installation Guide Windows
This guide discusses how to install and check for correct operation of the CUDA Development Tools on Microsoft Windows systems.

Installation Guide Mac OS X
This guide discusses how to install and check for correct operation of the CUDA Development Tools on Mac OS X systems.

Installation Guide Linux
This guide discusses how to install and check for correct operation of the CUDA Development Tools on GNU/Linux systems.

Programming Guides