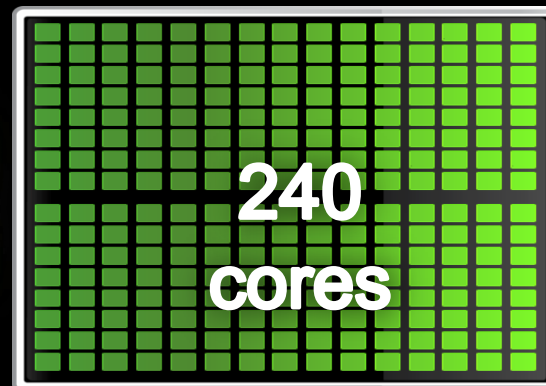
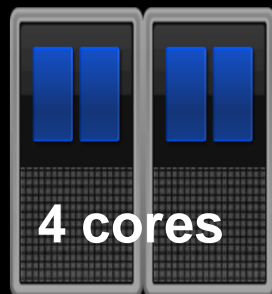
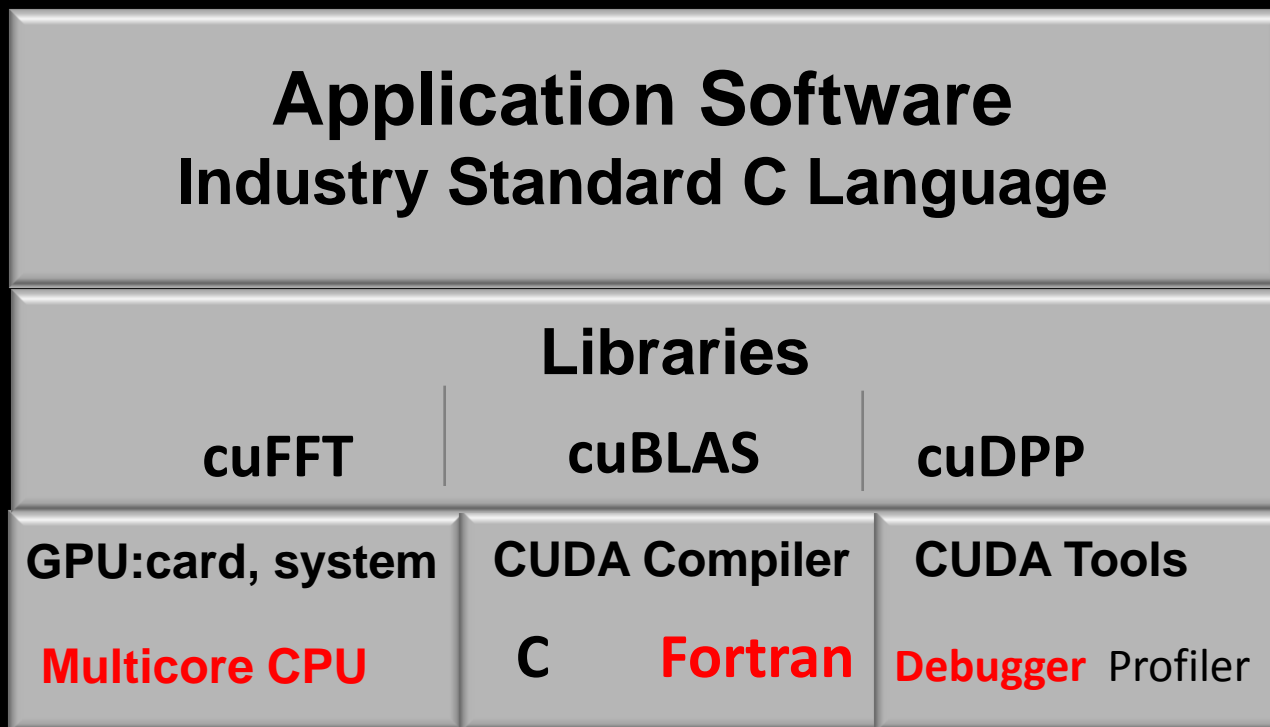


**nVIDIA®**

**CUDA Toolkit and Libraries**

**Scalable Parallel Programming with CUDA**

# CUDA Toolkit



# CUDA Compiler: nvcc



- Any source file containing CUDA language extensions (.cu) must be compiled with **nvcc**
- NVCC is a **compiler driver**
  - Works by invoking all the necessary tools and compilers like `cl`, `g++`, `cl`, ...
- NVCC can output:
  - Either C code (CPU Code)
    - To be compiled with the rest of the application using another tool
  - Or PTX object code directly
- **An executable with CUDA code requires:**
  - The CUDA core library (**cuda**)
  - The CUDA runtime library (**cuda**)

# CUDA libraries



- **CUDA includes 2 widely used libraries**
  - **CUBLAS: BLAS implementation**
  - **CUFFT: FFT implementation**
  
- **CUDPP (Data Parallel Primitives), available from [www.gpgpu.org/developer/cudpp](http://www.gpgpu.org/developer/cudpp):**
  - **Reduction**
  - **Scan (prefix sum)**
  - **Sort**

# CUBLAS



- **Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA driver**
  - Self-contained at the API level, no direct interaction with CUDA driver
- **Basic model for use**
  - Create matrix and vector objects in GPU memory space
  - Fill objects with data
  - Call sequence of CUBLAS functions
  - Retrieve data from GPU
- **CUBLAS library contains helper functions**
  - Creating and destroying objects in GPU space
  - Writing data to and retrieving data from objects

# Supported Features



- **BLAS functions**
  - **Single precision data:**
    - Level 1 (vector-vector  $O(N)$  )
    - Level 2 (matrix-vector  $O(N^2)$  )
    - Level 3 (matrix-matrix  $O(N^3)$  )
  - **Complex single precision data:**
    - Level 1
    - CGEMM
  - **Double precision data:**
    - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
    - Level 2: DGEMV, DGER, DSYR, DTRSV
    - Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
- **Following BLAS convention, CUBLAS uses column-major storage**

# Using CUBLAS

- Interface to CUBLAS library is in **cublas.h**
- Function naming convention
  - cublas + BLAS name
  - Eg., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error
- Helper functions:
  - Memory allocation, data transfer
- Implemented using C-based CUDA tool chain
  - Interfacing to C/C++ applications is trivial

# Calling CUBLAS from FORTRAN



- **Two interfaces:**
  - **Thunking** (define CUBLAS\_USE\_THUNKING when compiling fortran.c)
    - Allows interfacing to existing applications without any changes
    - During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory
    - Intended for light testing due to call overhead
  - **Non-Thunking** (default)
    - Intended for production code
    - Substitute device pointers for vector and matrix arguments in all BLAS functions
    - Existing applications need to be modified slightly to allocate and deallocate data structures in GPGPU memory space (using CUBLAS\_ALLOC and CUBLAS\_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS\_SET\_VECTOR, CUBLAS\_GET\_VECTOR, CUBLAS\_SET\_MATRIX, and CUBLAS\_GET\_MATRIX)

# SGEMM example



```
! Define 3 single precision matrices A, B, C
real , dimension(m1,m1)::   A, B, C
.....
! Initialize
.....
#ifdef CUBLAS
! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
! memory allocation on device and data movement)
  call cublas_SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#else
! Call SGEMM in host BLAS library
  call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
```

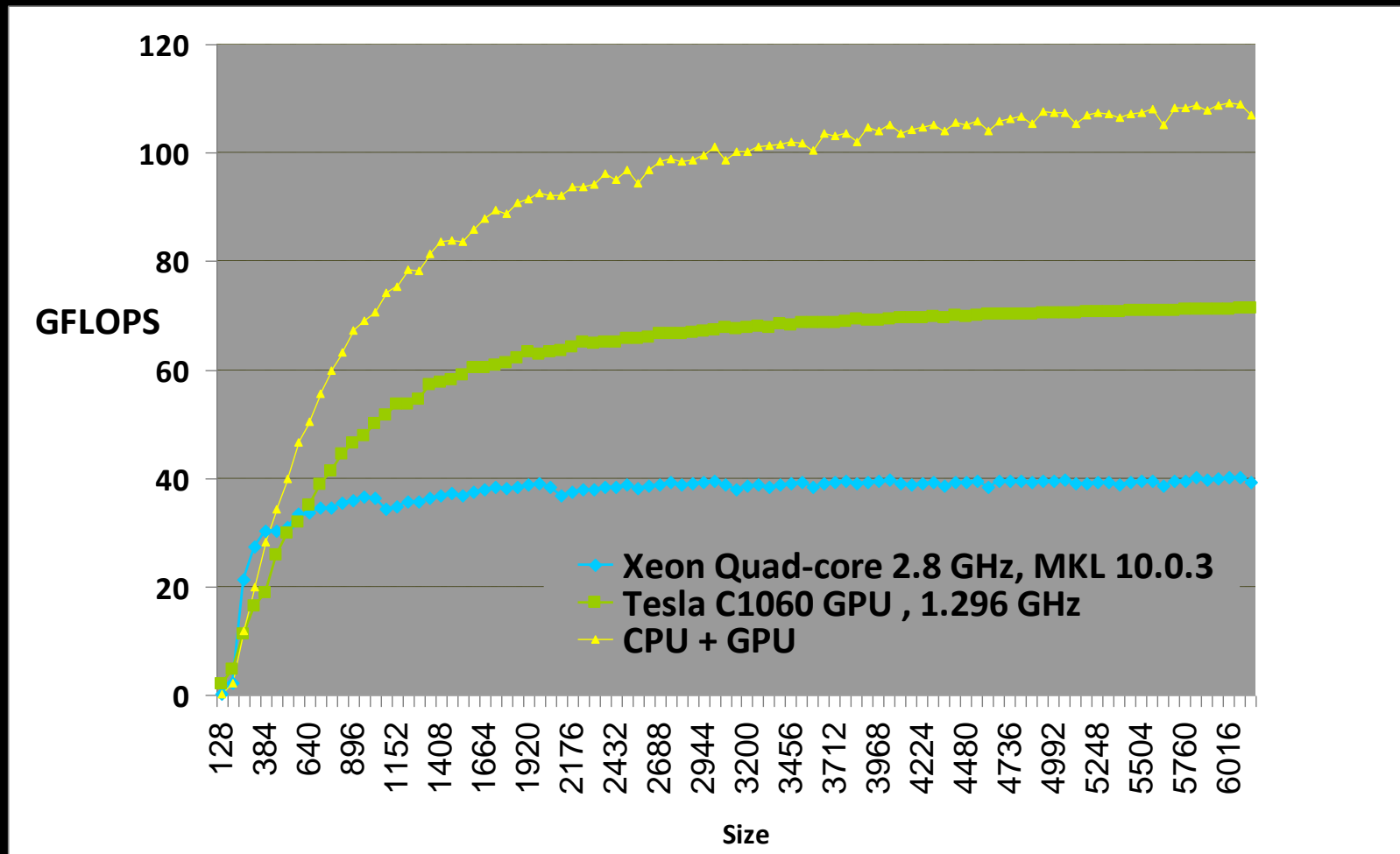
To use the host BLAS routine:

```
g95 -O3 code.f90 -L/usr/local/lib -lblas
```

To use the CUBLAS routine (fortran.c is provided by NVIDIA):

```
gcc -O3 -DCUBLAS_USE_THUNKING -I/usr/local/cuda/include -c fortran.c
g95 -O3 -DCUBLAS code.f90 fortran.o -L/usr/local/cuda/lib -lcublas
```

# DGEMM Performance



# CUFFT



- **The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm for efficiently computing discrete Fourier transform of complex or real-valued data sets.**
- **CUFFT is the CUDA FFT library**
  - **Provides a simple interface for computing parallel FFT on an NVIDIA GPU**
  - **Allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation**

# Supported Features



- **1D, 2D and 3D transforms of complex and real-valued data**
- **Batched execution for doing multiple 1D transforms in parallel**
- **1D transform size up to 8M elements**
- **2D and 3D transform sizes in the range [2,16384]**
- **In-place and out-of-place transforms for real and complex data.**

# Transform Types

- **Library supports real and complex transforms**
  - `CUFFT_C2C`, `CUFFT_C2R`, `CUFFT_R2C`
- **Directions**
  - `CUFFT_FORWARD` (-1) and `CUFFT_INVERSE` (1)
    - According to sign of the complex exponential term
- **Real and imaginary parts of complex input and output arrays are interleaved**
  - `cufftComplex` type is defined for this
- **Real to complex FFTs, output array holds only nonredundant coefficients**
  - $N \rightarrow N/2+1$
  - $N_0 \times N_1 \times \dots \times N_n \rightarrow N_0 \times N_1 \times \dots \times (N_n/2+1)$
  - For in-place transforms the input/output arrays need to be padded

# More on Transforms



- For 2D and 3D transforms, CUFFT performs transforms in row-major (C-order)
- If calling from FORTRAN or MATLAB, remember to change the order of size parameters during plan creation
- CUFFT performs un-normalized transforms:  
$$\text{IFFT}(\text{FFT}(A)) = \text{length}(A) * A$$
- CUFFT API is modeled after FFTW. Based on plans, that completely specify the optimal configuration to execute a particular size of FFT
- Once a plan is created, the library stores whatever state is needed to execute the plan multiple times without recomputing the configuration
  - Works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources

# Code example: 2D complex to complex transform

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
...
/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Note:
   Different pointers to input and output arrays implies out of place transformation
*/

/* Destroy the CUFFT plan. */
cufftDestroy(plan);

cudaFree(idata), cudaFree(odata);
```