Architect of an Open World™

Integrating HPC and GPU processors

Experience with NVIDIA Tesla

LIBERATE IT

Contents - Introduction

Motivation

Frequency scaling is over

⇒ **We are now scaling cores**

- Scaling cores in a system
  - Memory wall continues to get worse
- Core scale out (increase server numbers)
  - Network bandwidth continues to increase
  - Network latency is limited by distance

☞ *Specialized Massively parallel computers*
have lost the economic argument
against the advance of ***commodity technology***

Why using NVIDIA GPU for computing ?

- GPU = Graphics Processing Unit (= device)
  - Chip in 3D computer video card =>
    GPU is a commodity component
- NVIDIA GPU is massively multi-threaded many cores
  - **Up to 240 threads executed in parallel**
  - **Up to 30720 concurrent threads in flight**
- **NVIDIA GPU is fast**
  - Theoretical peak performance:
    - 1 TeraFlops in single precision
    - 85 GigaFlops in double precision
  - Memory access peak bandwidth: 102GB/s

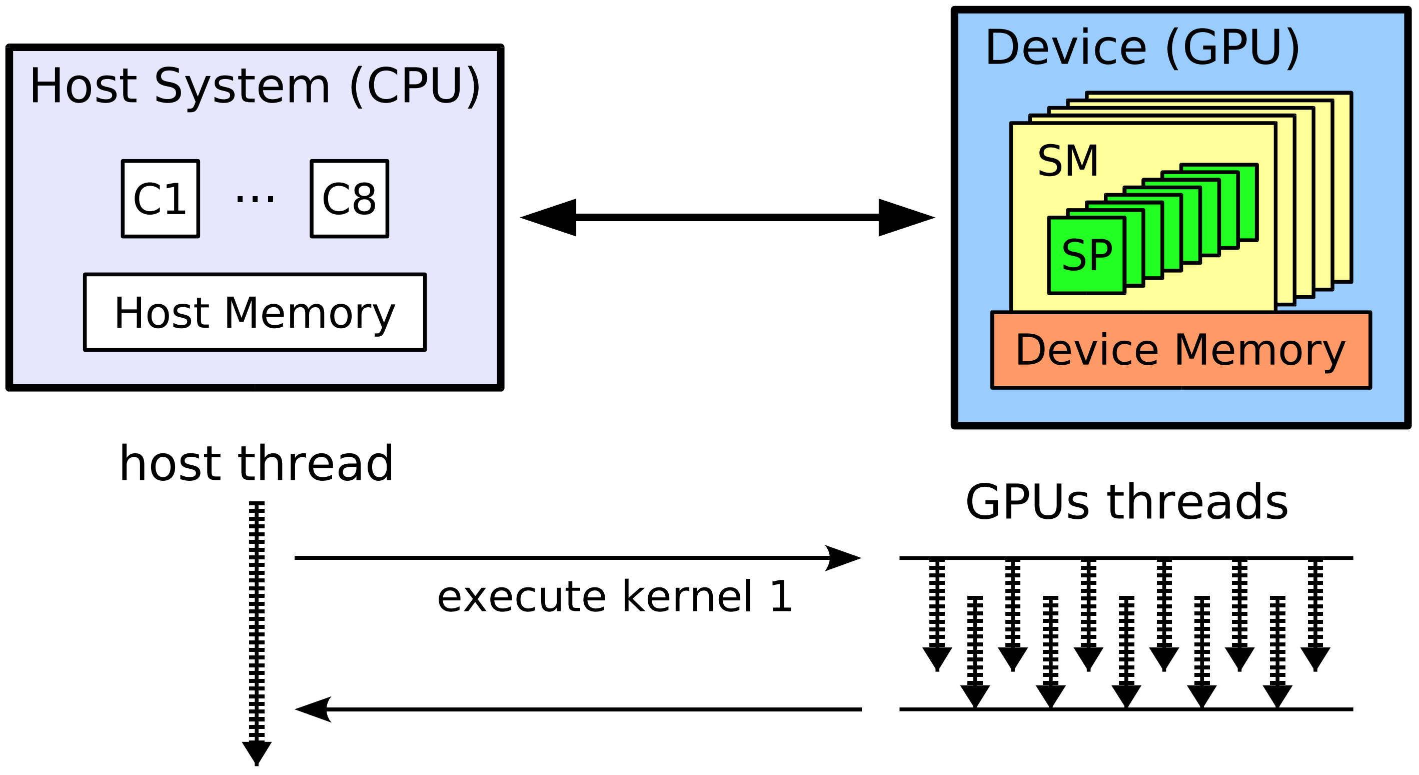
How to use NVIDIA GPU power for computing ?

CUDA = Compute Unified Device Architecture

- CUDA is a scalable programming model and a software environment for parallel computing
  - Extension to C/C++ environment
  - Heterogeneous serial-parallel programming model
  - Enable general-purpose GPU computing
  - Expose the computational horsepower of NVIDIA GPUs

$\Rightarrow$ NVIDIA GPU computing with CUDA brings parallel computing to the masses

Introduction to GPU execution model



kernel = function called from the host that runs on the device

8 Scalar Processor cores (SP) per Streaming Multiprocessor (SM)

Introduction to CUDA programming model

A Highly Multi-threaded Co-processor

- The GPU is a compute device
  - serves as a co-processor for the host CPU
  - has its own device memory on the card
  - has a set of processor cores organized hierarchically
- The GPU is Highly Multi-threaded
  - runs a single code (kernel) in many threads
  - executes many threads in parallel
  - uses multiple active threads per compute unit
- GPU threads are extremely lightweight
  - thread creation and context switching are essentially free
- GPU expects 1000's of threads for full utilization

Contents - Introduction

- Why use GPU for computing ?
- **Hardware items for GPU computing**
- Overview of Hardware Architecture
- Software Components : CUDA from NVIDIA

Overview of NVIDIA Hardware Product

- Chip series G8x, G9x or GT200/T10
- CUDA enabled products
  - NVIDIA TESLA - GPU solution for HPC (No video output)
    - Computing processor (board): C870, C1060
    - Deskside computing system: D870
    - 1U GPU Computing system: S870, S1070
  - NVIDIA Quadro - GPU solutions for 3D professional
    - Quadro FX 3700, Quadro FX 1700, ...
    - Quadro FX 5600, Quadro FX 4600, ...
  - NVIDIA GeForce - GPU for 3D on desktop
    - GeForce GTX 280, Ge Force GTX 260
    - GeForce 9800*, GeForce 9600*, ...
    - GeForce 8800*, GeForce 8600*, ...

NVIDIA Tesla S1070

- 4 Teraflops peak in 1U
- **4 x GPUs** -- model **GT200/T10**
- 120 Streaming Multiprocessors (30 per GPU)
- 960 scalar processor cores at 1.44GHz (240 per GPU)
- **IEEE754 single and double precision**
- 16GB of memory (4x**4GB**), 512-bit GDDR3 at 800MHz
- **2 Host Interface Card (HIC) PCIe 2.0 x16 (8GB/s)**
- 700 Watts (own power supply)

Compute capability of NVIDIA GPUs

Different GPU chip series used in different products
⇒ **With different features in different GPUs**

The **Compute Capability** level of a GPU determines
- Computing features of the GPUs
- Some hardware characteristics of a GPUs

compute capability 1.(n+1)
supersedes compute capability 1.n

☞ *The Software CUDA can interrogate the compute device (GPU) to determine its compute capability*

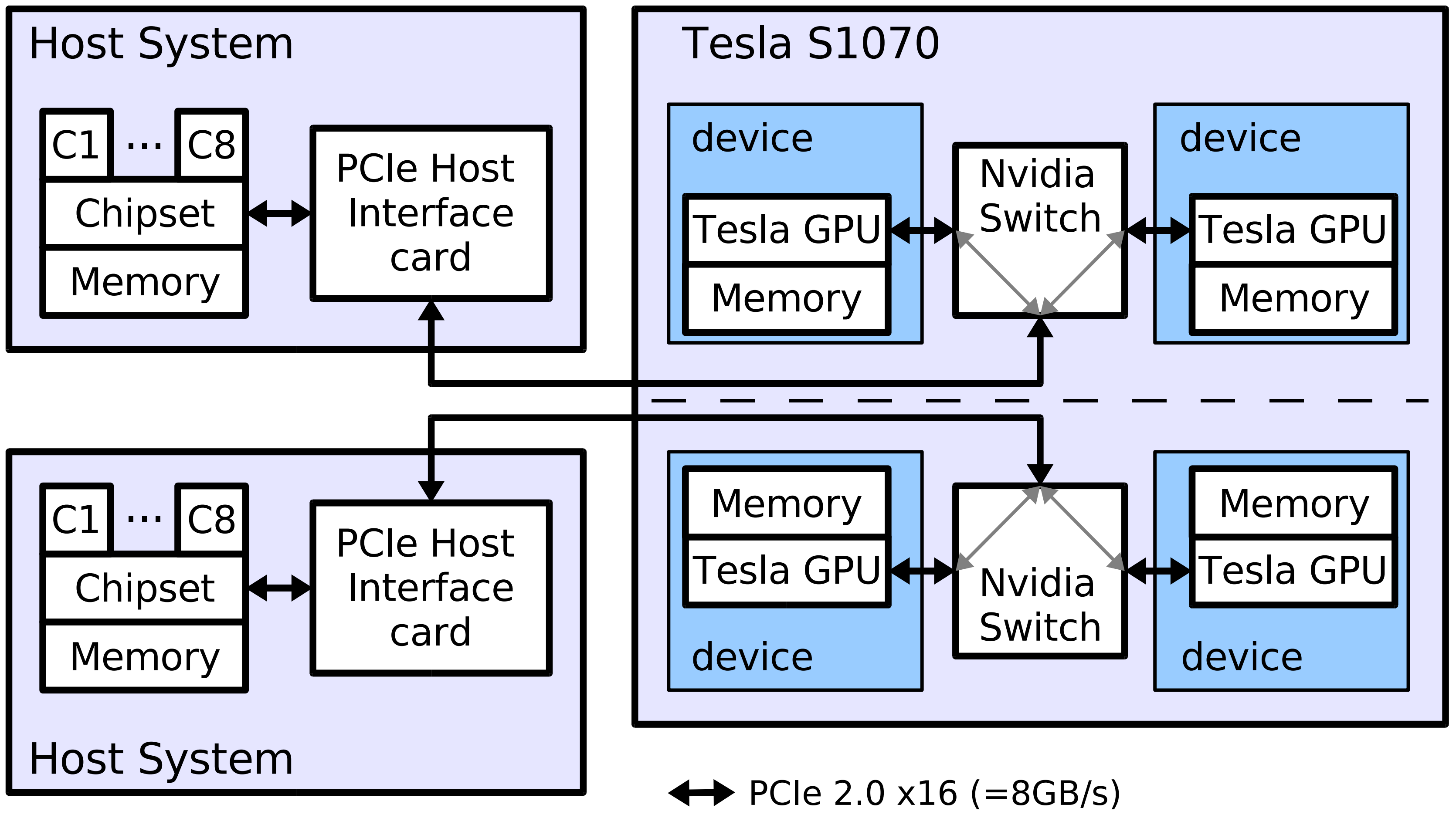
Product with latest compute capability 1.3:
- NVIDIA Tesla S1070, Tesla C1060
- NVIDIA GeForce GTX280, GeForce GTX260

Support for double-precision floating point numbers

Contents - Introduction

System Architecture - block diagram

Connection between Host System and Tesla S1070



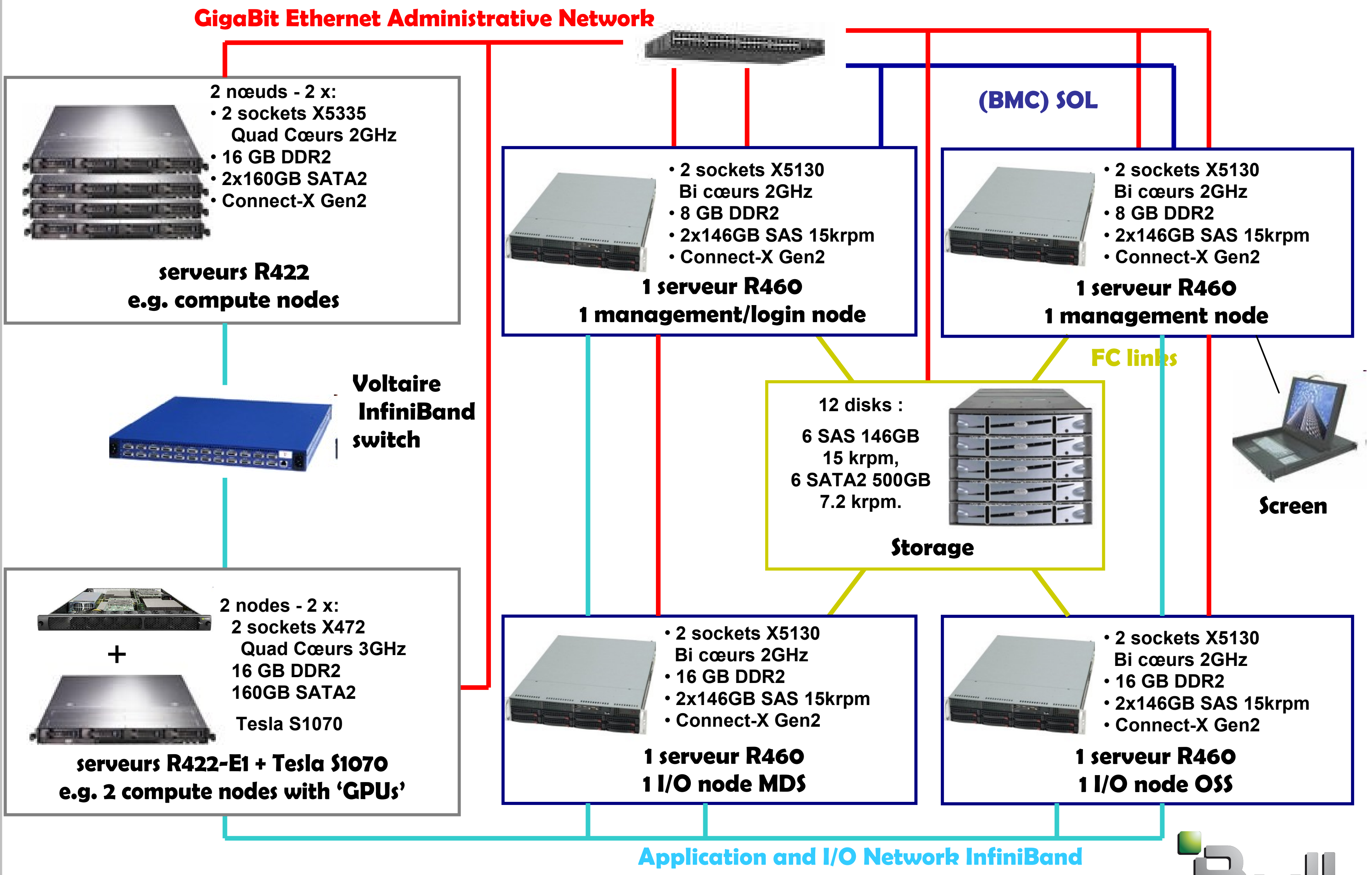
Bull targeted servers for GPU computing

Bull NovaScale for HPC with

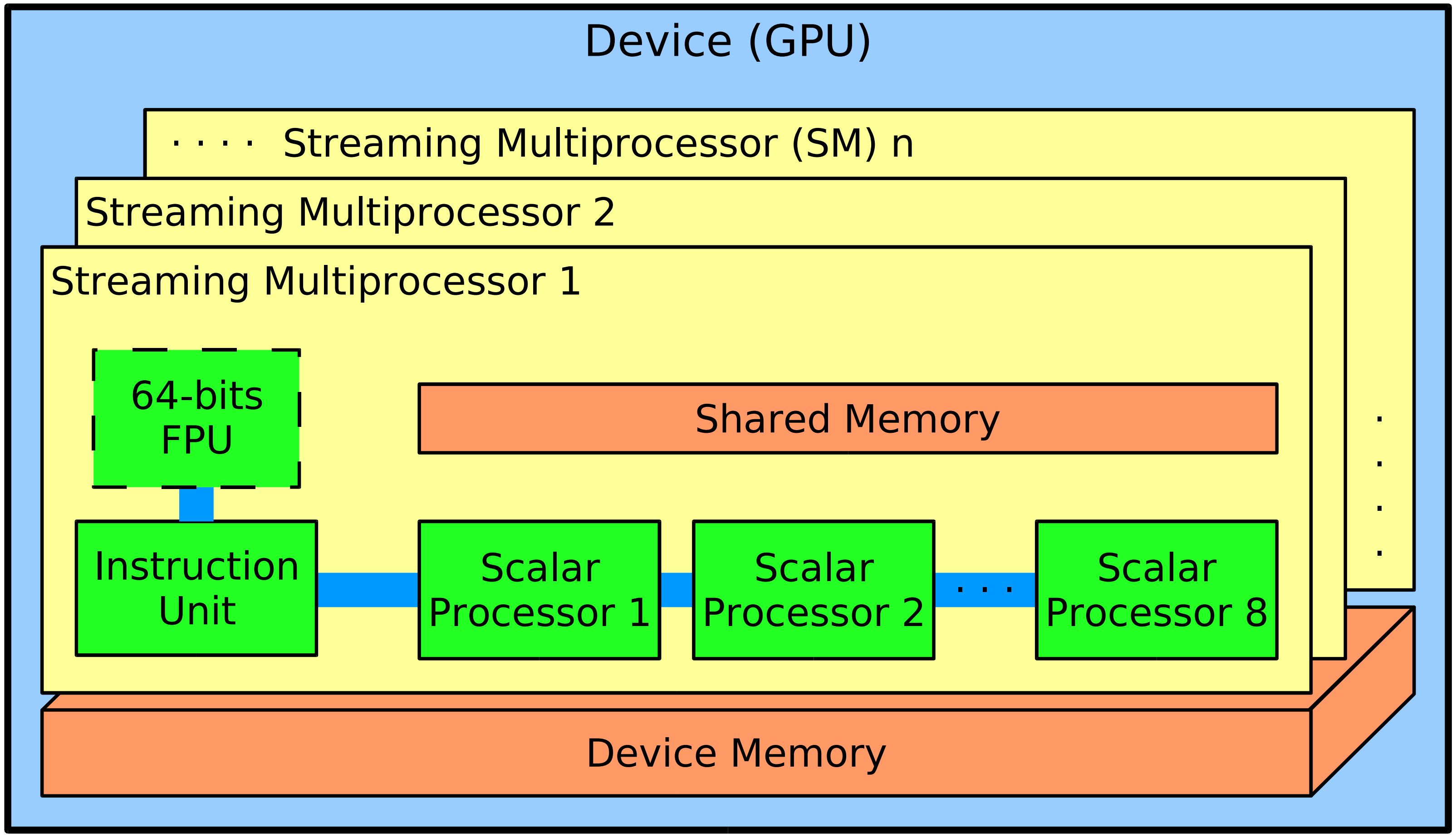
2 x dual core Intel® Xeon® (5200) at up to 3.4GHz OR
2 x quad core Intel® Xeon® (5400) at up to 3.2GHz

- **NovaScale R421-E1**
  - Connect up to 2GPUs of half of a Tesla S1070
- **NovaScale R422-E1**
  - 2 x servers in 1U
  - Connect up to 2GPUs of half of a Tesla S1070 per server
- **NovaScale R425**
  - 2 x bus slots PCIe x16 Gen2
  - **Connect up to 4GPUs of a Tesla S1070** OR
    Connect up to 2 Tesla C1060

Typical architecture



GPU architecture overview - block diagram



GPU architecture overview - Characteristics

- Device (GPU) contains:
  - a device memory of type GDDR3,
  - a set of Streaming Multiprocessors (SM)
- A Streaming Muliprocessor contains:
  - one Instruction Unit,
  - 8 x 32-bits Scalar Processor cores (SP),
  - one 64-bit Floating Point Unit (only on GT200),
  - 16KB of shared memory, local to each SM.
    It means, shared only between Scalar Processor cores of the same Streaming Multiprocessor.

The SPs of an SM work synchronously on the same instruction **=>**
SIMT = Single Instruction Multiple Thread

Peak Performance of Tesla S1070

NVIDIA Tesla S1070: 4 GPUs with

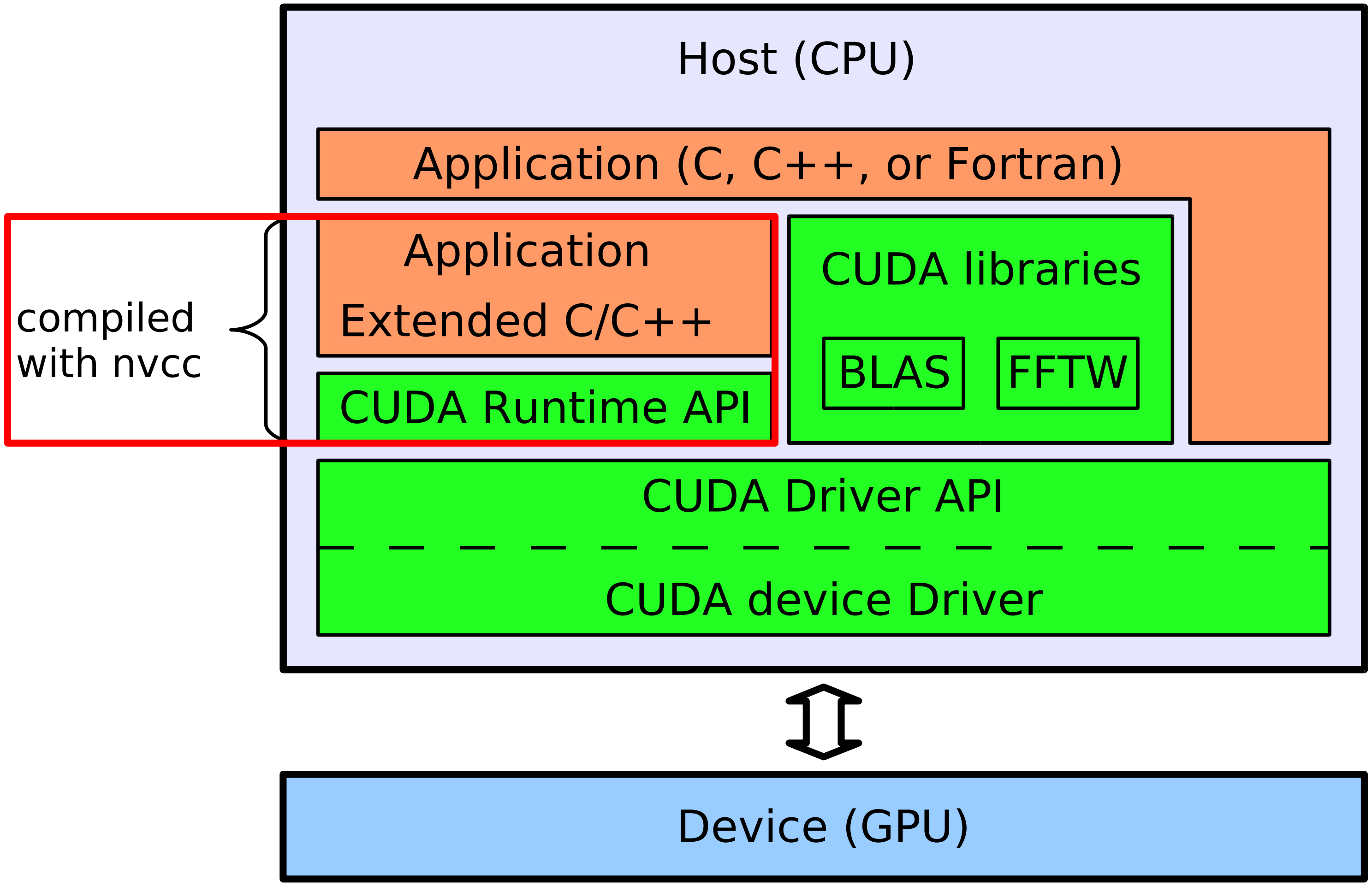
- 30 Streaming Multiprocessors with one FMAD (2 op) Double Precision each cycle per GPU,
- 240 (=30x8) Scalar Processor cores with one FMAD (2 op) and one FMUL (1 op) Single Precision each cycle per GPU,
- GPU Frequency 1.44 GHz,
- Memory GDDR3 dual channel, 512-bits wide at 800MHz

- Single Precision: **4147 GFlops/s**
  = 4 GPU x 240 SP x (2 + 1) x 1.44 GHz
- Double Precision: **345 GFlops/s**
  = 4GPU x 30 SM x 2 x 1.44 GHz
- Device Memory Bandwidth: **409,6 GB/s**
  = 4 GPU x 2 channel x (512 bits / 8) x 800MHz

Contents - Introduction

- Why use GPU for computing ?
- Hardware items for GPU computing
- Overview of Hardware Architecture
- **Software Components : CUDA from NVIDIA**

CUDA software Layer



CUDA Software Components

- CUDA Driver
  - Nvidia driver with CUDA support for Linux 64-bit
- CUDA Toolkit
  - nvcc Extended C/C++ compiler
  - **CUDA FFT** and **CUDA BLAS** libraries
  - gdb debugger for GPU
  - Emulation libraries (for Emulation of GPU code on CPU)
  - CUDA Runtime API library
  - CUDA programing manual
- CUDA Developer SDK
  - Set of examples with source code.
- CUDA Profiler

What is the CUDA Runtime API ?

- An Application Programing Interface (API) to use the device/GPU from the host/CPU code.
- Extensions to C/C++ language to write and call device kernel.
- A compiler of the extended C/C++ language (nvcc).
- Shared runtime libraries to link with CUDA code.
  - Set of shared libraries to use the device
  - Set of shared libraries to emulate the device on the CPU.

Application guideline

- Identify Hot Spot (75% of computational time)
  => Write GPU code only for that part
- Single precision performance is more than 8x better than double precision.
- Keep data on GPU memory.
- Prefer structure of arrays than an array of structure.

- Double precision application needs hybrid code

Ex.: Hybrid DGEMM (Matrix product double precision)
40% on 4 cores using OpenMP and
60% on one GPU

Architect of an Open World™

LIBERATE IT

Example: Compute of f(x,y) on a 2D domain

```
void comp2Df( int n, float *tF ) {
  float *d_tF;
  dim3 blkDim, grdDim;
  int memSize = n * n * sizeof(float);
  cudaMalloc( (void**)&d_tF, memSize );
  blkDim.x = blkDim.y = 16;
  grdDim.x = (n + 15)/blkDim.x;
  grdDim.y = (n + 15)/blkDim.y;
  k_comp2Df<<<grdDim,blkDim>>>(n, d_tF)
  cudaMemcpy( tF, d_tF, memSize, cudaMemcpyDeviceToHost);
  cudaFree(d_tF); }
```

Allocate memory on device

blkDim.x × blkDim.y $\leqslant$ 512

Call CUDA Kernel

Copy memory back to host

Free memory

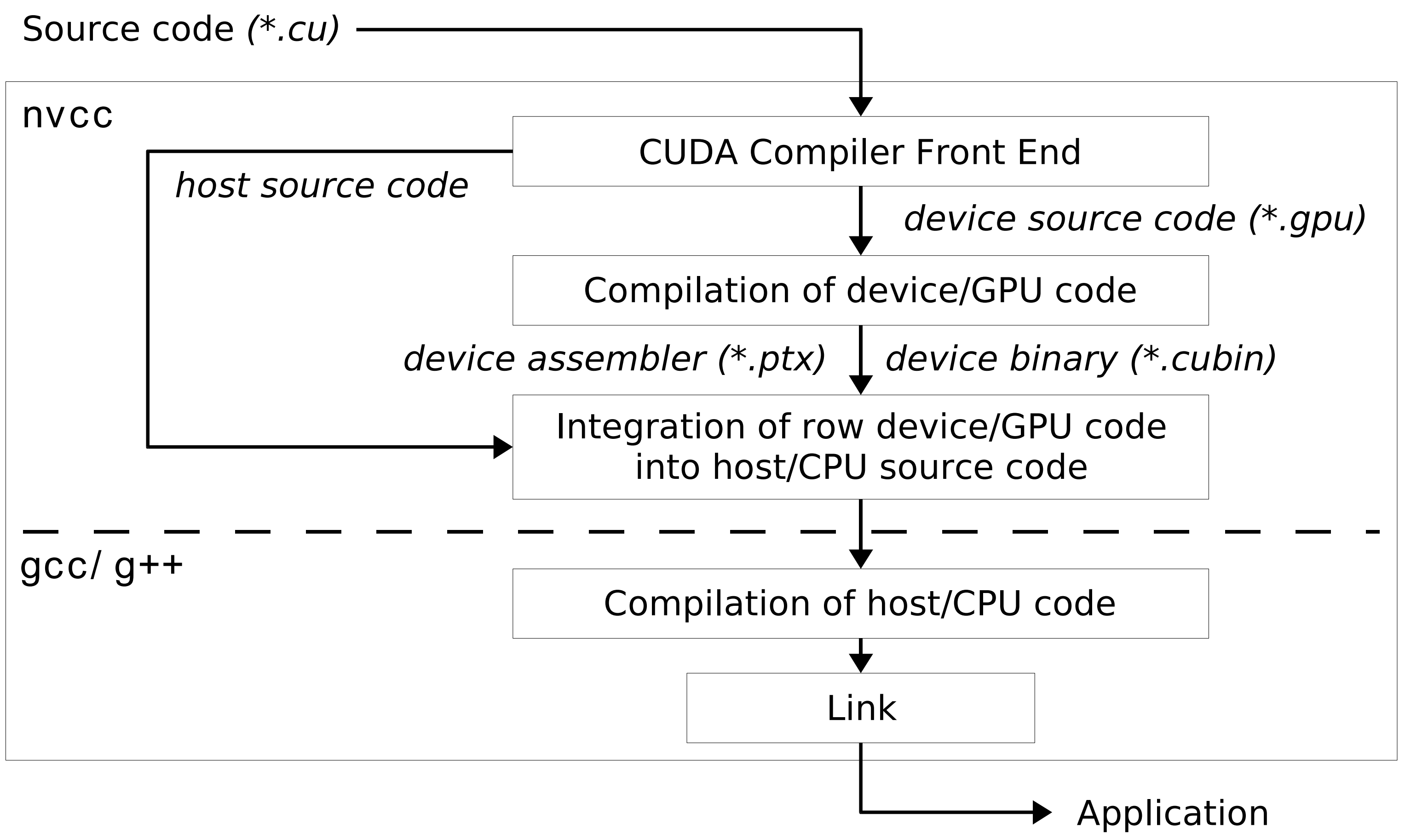
CUDA kernel

```
__global__ void k_comp2Df( int n, float *tF ) {
  float2 p;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if( (i < n) && (j < n) ) {
    p.x = (2.0 / (n - 1.0)) * i - 1.0;
    p.y = (2.0 / (n - 1.0)) * j - 1.0;
    tF[n * j + i] = f( p.x, p.y );
} }
```

Function on device

```
__host__ __device__ float f( float x, float y ) {
  return expf(-x*x-y*y); }
```

Compilation flow of CUDA code with nvcc


Source code *(*.cu)*
nvcc
CUDA Compiler Front End
host source code
device source code (.gpu)*
Compilation of device/GPU code
device assembler (.ptx)*
device binary (.cubin)*
Integration of row device/GPU code into host/CPU source code
gcc/ g++
Compilation of host/CPU code
Link
Application