JÜLICH
FORSCHUNGSZENTRUM

Supercomputing Centre Jülich (JSC)

Using JURECA's GPU Nodes

Willi Homberg

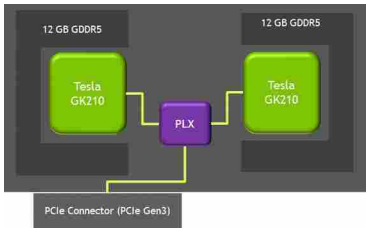
Introduction to the usage and programming of supercomputer resources in Jülich
22-23 May 2017

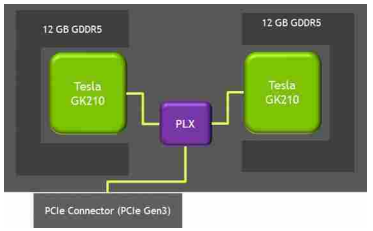
Outline

- JURECA'S GPU partition
- Why having a GPU partition ?
  - CPU vs. GPU architecture
  - GPU computing and processing flow
  - NVIDIA Tesla K80
- APIs and libraries
- Compile and build
- Resource allocation and job execution
- Profiling and performance analysis
- Guidance and support

JURECA's GPU Partition

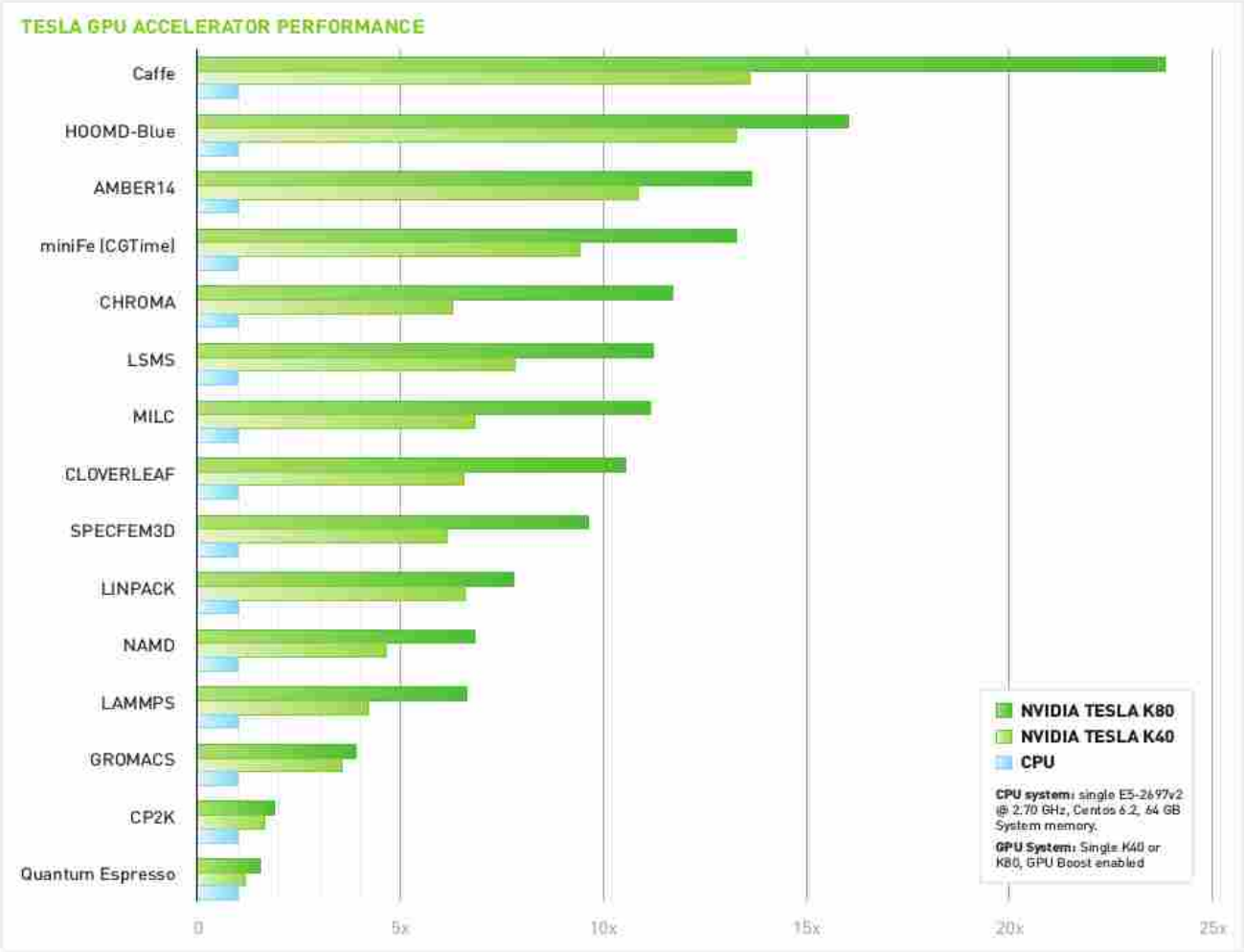
- 75 compute nodes equipped with
  - two **Intel Xeon E5-2680 v3 Haswell CPUs**
  - 2x 12 cores, 2.5 GHz
  - SMT, AVX 2.0
  - 960 GFlop/s (DP) per node
  - 128 GiB DDR4 memory
  - **plus**
  - two **NVIDIA K80 GPUs** four visible devices per node
  - 2x 4992 CUDA cores, 810-875 MHz
  - 3,740 GFlop/s (DP) per node
  - 2x 24 GiB GDDR5 memory





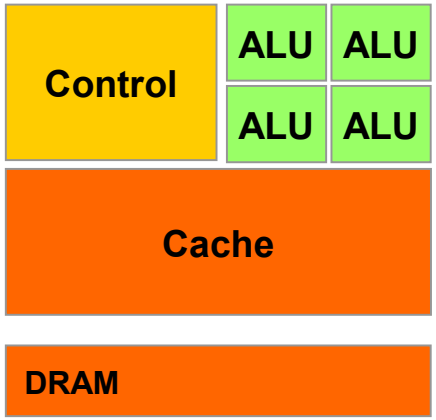
Why having a GPU Partition on JURECA ?

- GPU-computing, i.e. CPU and graphic processor are jointly used to accelerate scientific and technical applications
- Compute-intensive parts of an application are transferred to the GPU while the remaining code runs on the CPU as usual
- From a user's point of view, in many cases the time-to-solution can be considerably reduced
- Take over the task of the decommissioned GPU cluster JUDGE



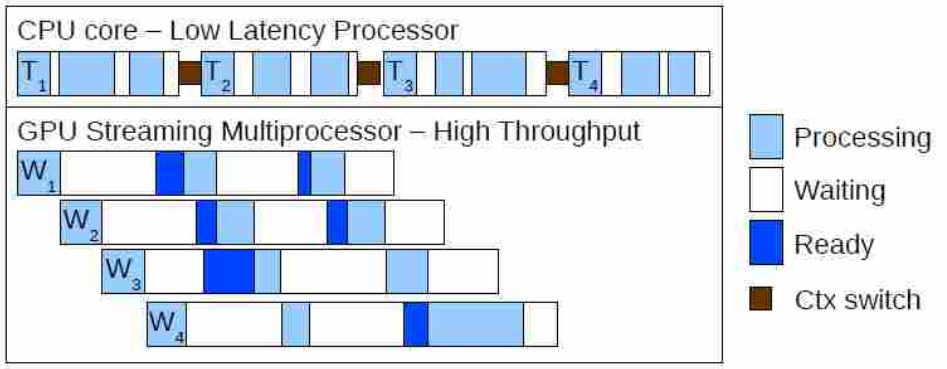
© NVIDIA Tesla K80 data sheet

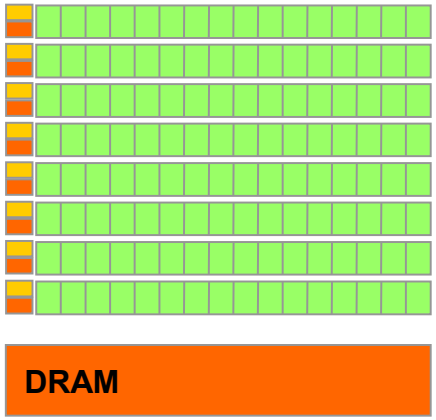
CPU vs. GPU: Comparison of Architectures



CPU

- Optimized for low-latency access to cached data sets for each thread
- Control logic for out-of-order and speculative execution





GPU

- Optimized for data-parallel throughput computation
- GPU architecture hides latency with computation from other thread warps (bundle of threads)
- Architecture tolerant of memory latency
- More transistors dedicated to computation

GPU Computing

Fig.: Nvidia

Kepler GPU (GK110):

- Each green square = single FPU
- Each FPU **(about 2700)** available for a different thread
- Overall, GK110 can handle **more than 30000 threads** simultaneously…
- …and even better, each program can send **billions of threads** to the GPU!

- GPU programming: **Thinking in large arrays of threads**
  - Proper organization of threads incl. data sharing very important
- Many APIs for many different programming languages available:
  - CUDA (only NVIDIA; e.g., runtime C++ API)
  - OpenCL (independent of hardware platform, also for CPUs)
  - OpenACC (for NVIDIA and AMD GPUs; based on compiler directives like OpenMP)

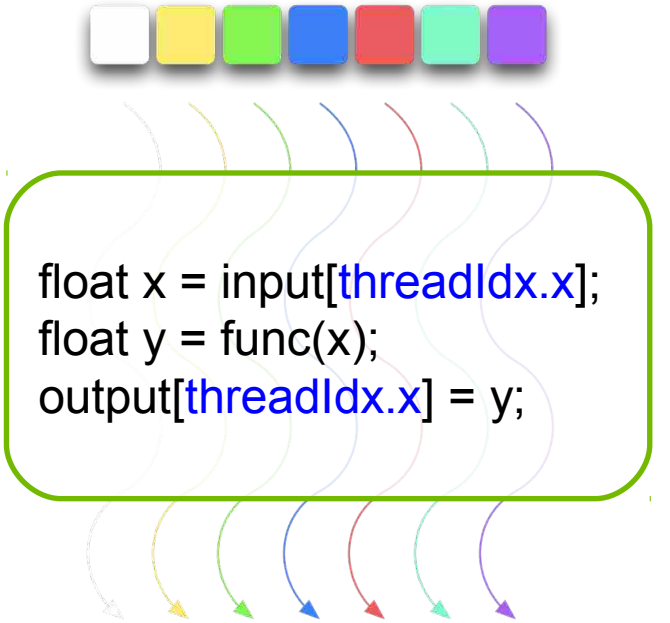
CUDA Kernels: Parallel Threads

- A kernel is a function executed on the GPU as an array of threads in parallel
  - executing a parallel portion of application
  - entire GPU executes kernel, many threads

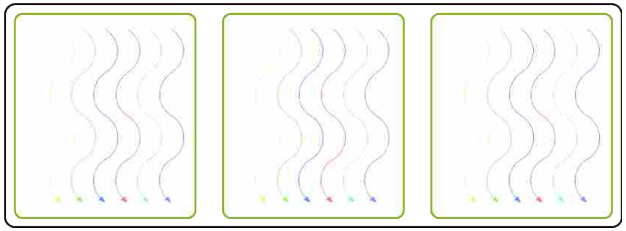
- All threads execute the same code, but can take different paths

- Each thread has an ID
  - select input/output data
  - control decisions

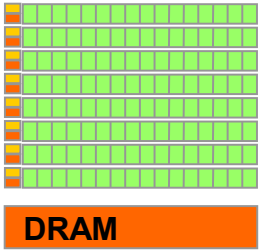
- CUDA threads:
  - lightweight
  - fast switching
  - 1000s execute simultaneously



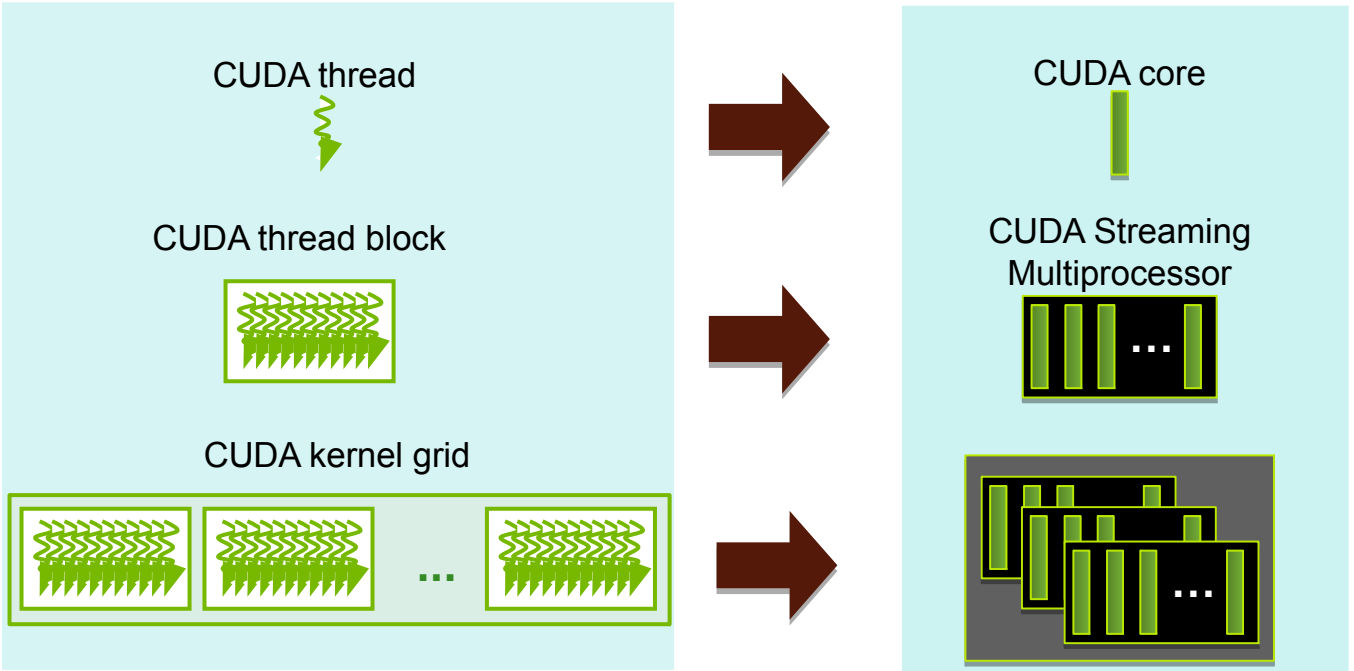
CUDA Kernel execution: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads



- Block can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



Scale Kernel

```
void scale(float alpha,
                    float* A,
                    float* C,
                    int m)
{
     int i = 0;
     for ( i=0; i<m; ++i)
          C[i] = alpha * A[i];
}
```

```
__global__ void scale(float alpha,
                    float* A,
                    float* C,
                    int m)
{
     int i = blockDim.x*blockIdx.x+threadIdx.x;
     if ( i < m)
          C[i] = alpha * A[i];
}
```

On JURECA (Tesla K80): **Thread block dimension**

- Max. dim. of a block: 1024 x 1024 x 64
- Max. number of threads per block: 1024

Example:

// Create 3D thread block with 512 threads

dim3 blockDim(16, 16, 2);

On JURECA (Tesla K80): **Grid dimension**

- Max. dim. of a grid: 2147483647 x 65535 x 65535

Example:

// Dimension of problem: nx x ny = 1000 x 1000

dim3 blockDim(16, 16) // Don't need to write z = 1

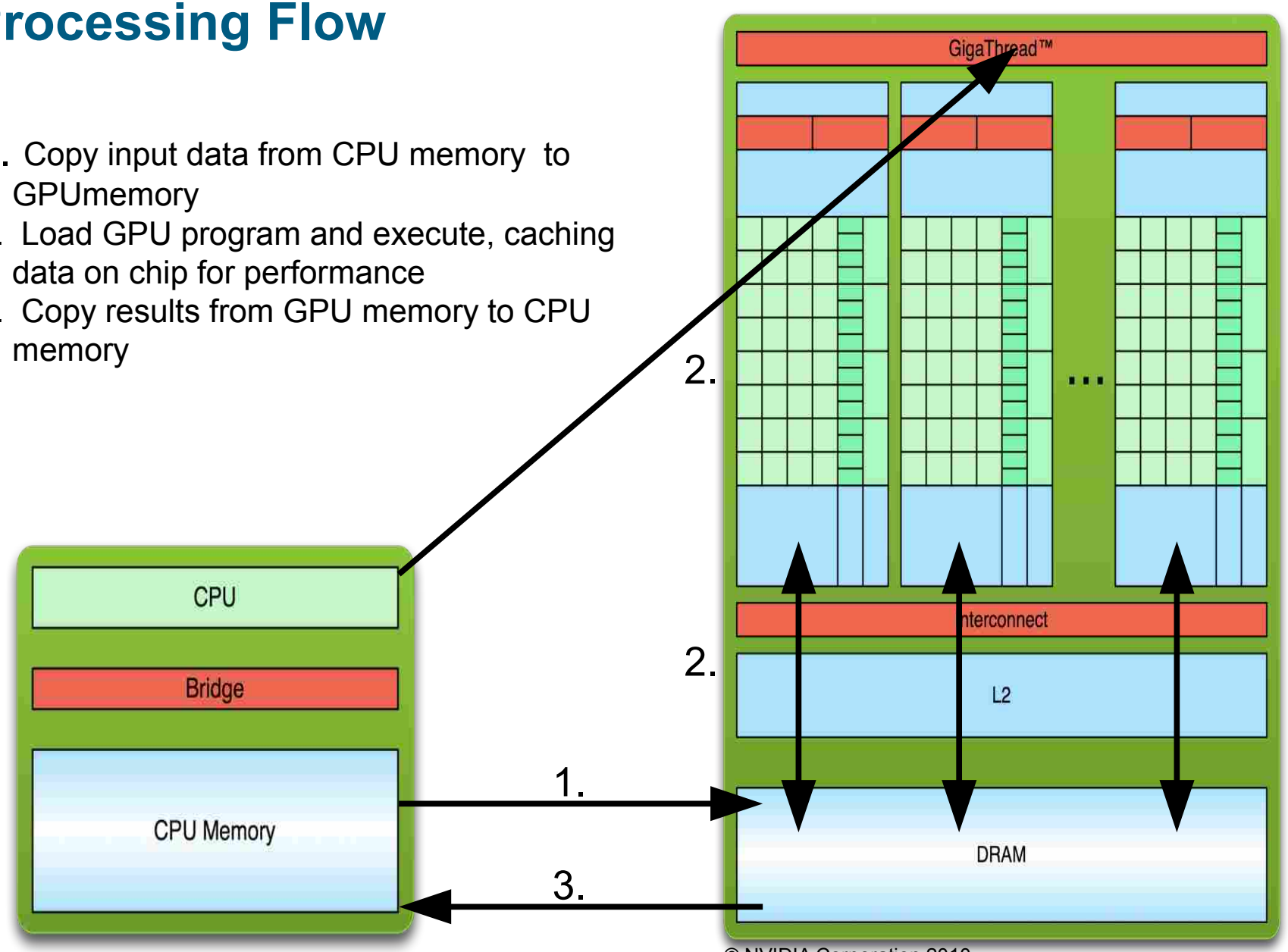
int gx = (nx % blockDim.x==0) ? nx / blockDim.x : nx / blockDim.x + 1

int gy = (ny % blockDim.y==0) ? ny / blockDim.y : ny / blockDim.y + 1

dim3 gridDim(gx, gy);

Processing Flow

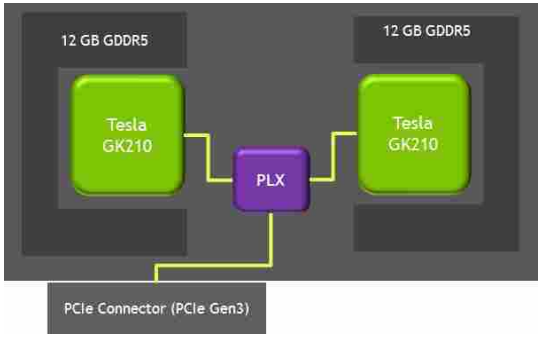
1. Copy input data from CPU memory to GPUmemory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

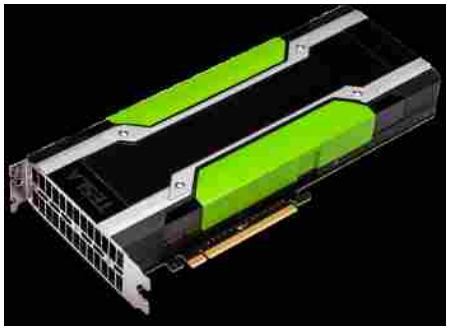


NVIDIA K80 GPU

- TESLA K80 GPU Accelerator board specification

  http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf





TECHNICAL SPECIFICATIONS	Tesla K40	Tesla K80[1]
Peak double-precision floating point performance (board)	1.43 Tflops	1.87 Tflops
Peak single-precision floating point performance (board)	4.29 Tflops	5.6 Tflops
GPU	1 x GK110B	2 x GK210
CUDA cores	2,880	4,992
Memory size per board (GDDR5)	12 GB	24 GB
Memory bandwidth for board (ECC off)[2]	288 Gbytes/sec	480 Gbytes/sec
Architecture features	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers and workstations	Servers

GPU APIs and Libraries

APIs

- CUDA
  - Programming model developed by NVIDIA which only supports NVDIA devices; best software support
  - using FORTRAN code with CUDA possible via PGI compiler
- OpenACC
  - programming standard for CPU/GPU systems where the programmer identifies areas that should be accelerated using compiler directives (like in OpenMP); supported by PGI compiler
- OpenCL
  - open standard maintained by the (non-profit) technology organisation Khronos Group; generates portable code

NVIDIA GPU-accelerated libraries

- cuBLAS: GPU-accelerated version of the complete standard BLAS library
- cuSPARSE: collection of basic linear algebra subroutines used for sparse matrices
- cuFFT: CUDA Fast Fourier Transform Library
- cuRAND: Random Number Generation library
- Thrust: open source (template) library of parallel algorithms and data structures (sort, scan, transform, and reductions)
- CUSP: open source C++ library of generic parallel algorithms for sparse linear algebra and graph computations

https://developer.nvidia.com/gpu-accelerated-libraries

Using cuBLAS

Steps:

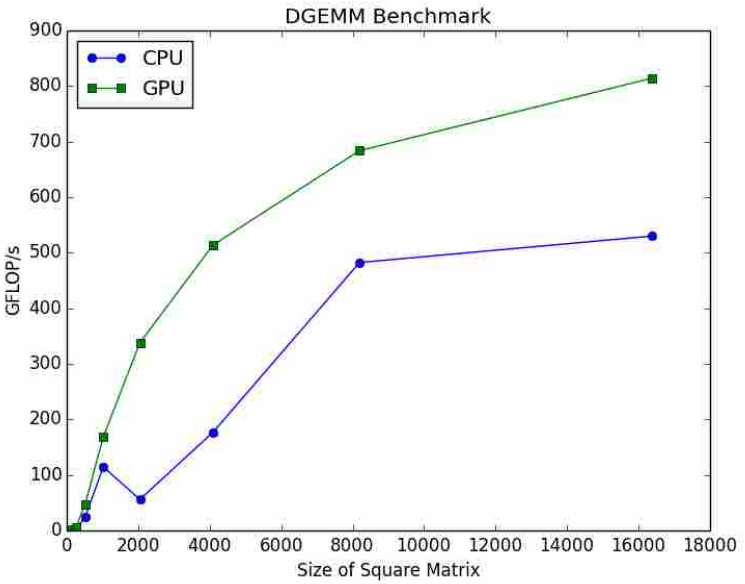
- Initialize
- Allocate memory on the GPU
- Copy data to GPU
- Call BLAS routine
- Copy results to Host
- Finalize

DDOT computation:

- status = cublasCreate(&handle)
- cudaMalloc((void**)&d_A, n * sizeof(d_A[0]))
- status = cublasSetVector(n, sizeof(A[0]), A, 1, d_A, 1);
- status = cublasDdot(handle, n, d_A, 1, d_B, 1, &res)
- status = cublasDestroy(handle);

DGEMM: DP Matrix Multiply

JURECA single node:
1 GPU vs.
2 CPUs, 24 cores



GPU Programming with CUDA

Programming model CUDA developed by NVIDIA

- GPU-accelerated libraries
- Tools for debugging, profiling and performance optimisations

Course at JSC:

PATC training course "GPU Programming with CUDA" (course no. 93/2017 in the training programme of Forschungszentrum Jülich)

- course announcement:

  https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2017/patc-gpu-cuda-2017.html

- GPU-accelerated computing drives current scientific research. Writing fast numeric algorithms for GPUs offers high application performance by offloading compute-intensive portions of the code to an NVIDIA GPU. The course will cover basic aspects of GPU architectures and programming. Focus is on the usage of the parallel programming language CUDA-C which allows maximum control of NVIDIA GPU hardware. Examples of increasing complexity will be used to demonstrate optimization and tuning of scientific applications.
- Topics covered:
  - Introduction to GPU/Parallel computing
  - Programming model CUDA
  - GPU libraries like CuBLAS and CuFFT
  - Tools for debugging and profiling
  - Performance optimizations
- course materials: http://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/EN/courses/2016/patc-gpu-cuda-2016.html

GPU Programming with OpenACC

Pragma/directive based progrogramming model OpenACC

- #pragma acc kernels in C
- !acc kernels … !acc end kernels in Fortran
- some additional control statements (copyin/copyout, vector, acc_init, acc data region)

Course at JSC:

“Introduction to GPU programming using OpenACC" (course no. 94/2017 in the training programme of Forschungszentrum Jülich)

- course announcement:

  http://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2017/gpu-openacc-2017.html?nn=717802

- GPU-accelerated computing drives current scientific research. Writing fast numeric algorithms for GPUs offers high application performance by offloading compute-intensive portions of the code to the GPU. The course will cover basic aspects of GPU architectures and programming. Focus is on the usage of the directive-based OpenACC programming model which allows for portable application development. Examples of increasing complexity will be used to demonstrate optimization and tuning of scientific applications.
- Topics covered:
  - Introduction to GPU/Parallel computing
  - Programming model OpenACC
  - Interoperability of OpenACC with GPU libraries like CuBLAS and CuFFT
  - Tools for debugging and profiling
  - Performance optimization
- course materials: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Services/Documentation/presentations/presentation-openacc_table.html?nn=362392

Programming with OpenCL

- primary goal of OpenCL
  - portability across a diverse set of computing devices including
    - CPUs, GPUs, and other accelerators

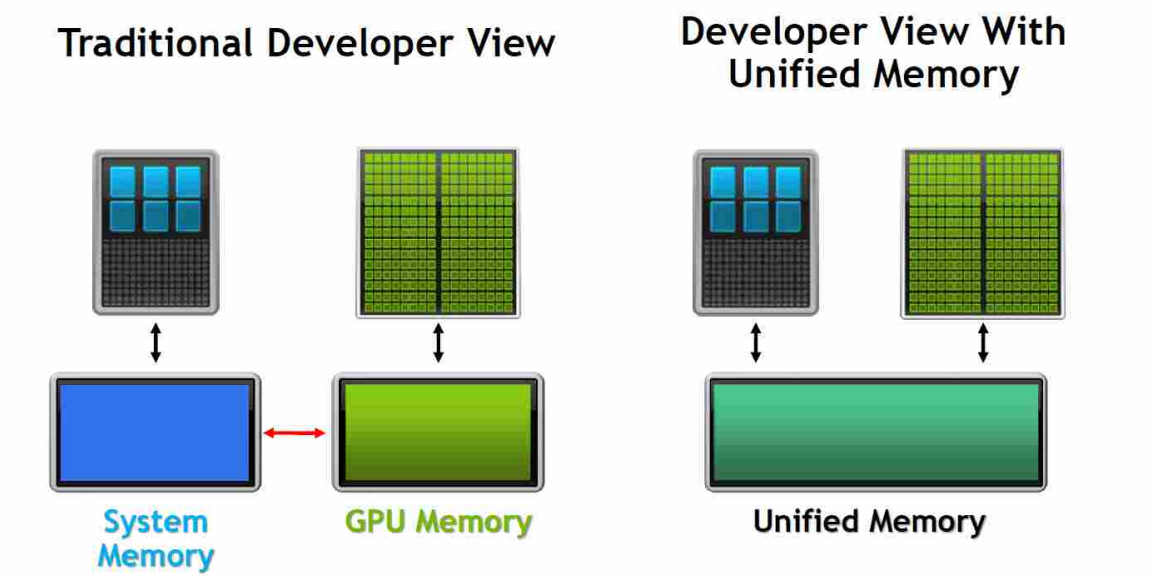
- Course at JSC:

“Vectorisation and Portable Programming using OpenCL" (course no. 95/2017 in the training programme of Forschungszentrum Jülich)

  - course announcement:

    http://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/EN/courses/2017/gpu-opencl-2017.html
  - OpenCL provides an open, portable C-based programming model for highly parallel processors. In contrast to NVIDIA's proprietary programming API CUDA, a primary goal of OpenCL is portability across a diverse set of computing devices including CPUs, GPUs, and other accelerators.
  - Topics covered:
    - Introduction to vectorisation
    - Programming model of OpenCL
    - Datatypes and OpenCL vectorisation features
    - Tuning for architectures like CPUs, accelerators (GPUs), and co-processors (Xeon Phi)
    - Heterogeneous multi-device programming
  - course materials:
    http://www.fz-juelich.de/ias/jsc/DE/Leistungen/Dienstleistungen/Dokumentation/Praesentationen/folien-opencl_table.html?nn=994290

Unified Memory: Access to Host and Device Data




```
void sortfile(FILE *fp, int N) {
  char *data;
  char *data_d;
  data = (char *)malloc(N);
  cudaMalloc( &data_d, N );

  fread(data, 1, N, fp);

  cudaMemcpy( data_d, data, N,
              cudaMemcpyHostToDevice);
  qsort<<<...>>>(data, N, 1, compare);

  cudaMemcpy( data, data_d, N,
              cudaMemcpyDeviceToHost);
  use_data(data);
  cudaFree(data_d); free(data); }
```

```
void sortfile(FILE *fp, int N) {
  char *data;

  cudaMallocManaged( &data, N );

  fread(data, 1, N, fp);

  qsort<<<...>>>(data, N, 1, compare);
  cudaDeviceSynchronize();

  use_data(data);
  cudaFree(data);              }
```

Unified Memory

Advantages:

- Unified Memory can be used in CPU and GPU code
- No need for explicit device allocation (cudaMalloc) or memory copies (cudaMemcpy)
- No need to fully understand data flow and allocation logic of application
- Incremental profiler driven acceleration → data movement ist just another optimisation

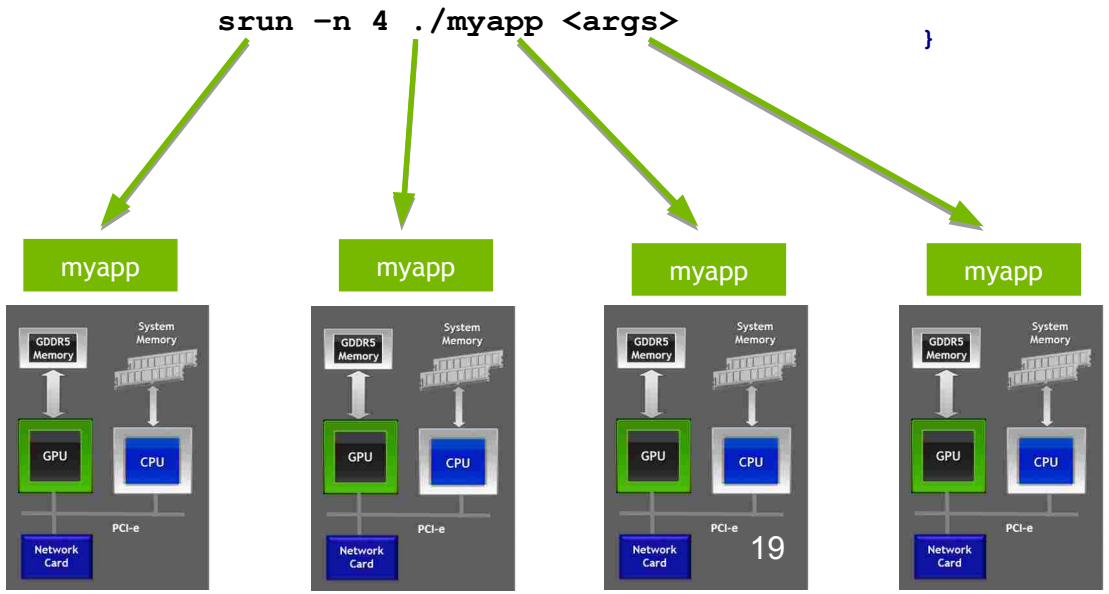
Implementation details:

- Unified Memory can only include heap and global data, no stack data
- Data is coherent only at kernel launch and sync points
- It is not allowed to access Unified Memory in host code while a kernel is running
    - doing so may result in a segmentation fault

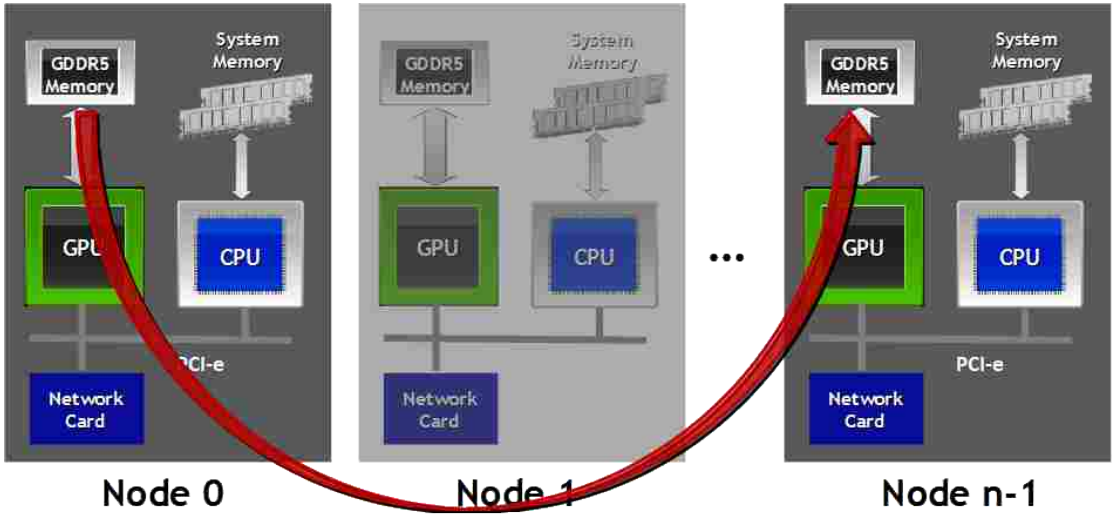
Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchange messages
    - Pt. 2 Pt.: e.g. MPI_Send, MPI_Recv
    - Collectives, e.g. MPI_Reduce
- Multiple implementations (open source and commercial)
  - Binding for C/C++, Fortran, Python, ...

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
      int myrank;
      /* Initialize the MPI library */
      MPI_Init(&argc,&argv);
      /* Determine the calling process rank */
      MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
      /* Call MPI routines like MPI_Send, MPI_Recv, ... */
      /* Shutdown MPI library */
      MPI_Finalize();
      return 0;
}
```



MPI + CUDA



With UVA and CUDA-aware MPI

```
//MPI rank 0
MPI_Send(s_buf_d,size,…);

//MPI rank n-1
MPI_Recv(r_buf_d,size,…);
```

No UVA and regular MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,…);
MPI_Send(s_buf_h,size,…);

//MPI rank n-1
MPI_Recv(r_buf_h,size,…);
cudaMemcpy(r_buf_d,r_buf_h,size,…);
```

Compile and Build Executable

Load the required GPU libraries:

module load defaults/GPU

CUDA:

- **GCC:** gcc/5.4.0

  nvcc uses g++ as default host compiler

- **wrapper nvcc_icpc:** icc/2016.4.258-GCC-5.4.0

  nvcc_icpc (or -ccbin=icpc) for use of icpc as host compiler

nvcc -o prog prog.cu

CUDA 8 is not compatible with Intel 2017 and PGI 17 compilers

cuBLAS:

CUBLASFLAGS = -I${CUDA_HOME}/include -L${CUDA_HOME}/lib64 -lcublas -lcudart

icc -o prog prog.cpp $(CUBLASFLAGS)

OpenACC:

- **PGI** compiler

module load PGI

pgcc -fast -acc -ta=tesla -Minfo=all -o prog prog.c

OpenCL:

- OpenCL version 1.2

INC=-I${CUDA_HOME}/include

LIB=-L${CUDA_HOME}/lib64

gcc -o prog prog.c -lOpenCL $(INC) $(LIB)

Resource Allocation and Job Execution

- Partitions: gpus and develgpus
  - -p gpus (or --partition gpus) or
  - -p develgpus (or --partition develgpus)
  - number of requested GPUs: --gres=gpu:X (1 <= X <= 4)
- Job scripts examples:
  - Example 1:
    - MPI application starting 96 tasks on 4 nodes using 24 CPUs per node and 4 GPUs per node
  - Example 2:
    - 4 (independent) job steps of a GPU program running on one node using one CPU thread and one GPU device each
- Command line commands on login node:
  - salloc --partition=develgpus –gres=gpu:N
  - srun –forward-x –cpu_bind=none –pty /bin/bash -i

    or sbatch <job-script>
- job status: squeue -u <userid>
- status of GPUs: nvidia-smi

Example 1

```
#!/bin/bash -x
#SBATCH --nodes=4
#SBATCH --ntasks=96
#SBATCH --ntasks-per-node=24
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=gpus

#SBATCH --gres=gpu:4

srun ./gpu-prog
```

Example 2

```
#!/bin/bash -x
#SBATCH --nodes=1
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:20:00
#SBATCH --partition=gpus
#SBATCH --gres=gpu:4

srun --exclusive -n 1 --gres=gpu:1 \
          --cpu_bind=map_cpu:0 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 \
          --cpu_bind=map_cpu:6 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 \
          --cpu_bind=map_cpu:12 ./gpu-prog &
srun --exclusive -n 1 --gres=gpu:1 \
          --cpu_bind=map_cpu:18 ./gpu-prog &
wait
```

Debugging, Profiling, and Performance Analysis

- You can only improve what you measure
  - Need to identify:
    - Hotspots: Which function takes most of the run time?
    - Bottlenecks: What limits the performance of the Hotspots?
- Manual timing is tedious and error prone
  - Possible for small application like matrix multiplication
  - Impractical for larger/more complex application
- Access to hardware counters (PAPI, CUPTI)

CUDA tools:

- cuda-memcheck to detect invalid memory accesses
- Nsight EE to debug a CUDA Program
- NVIDIA visual profiler

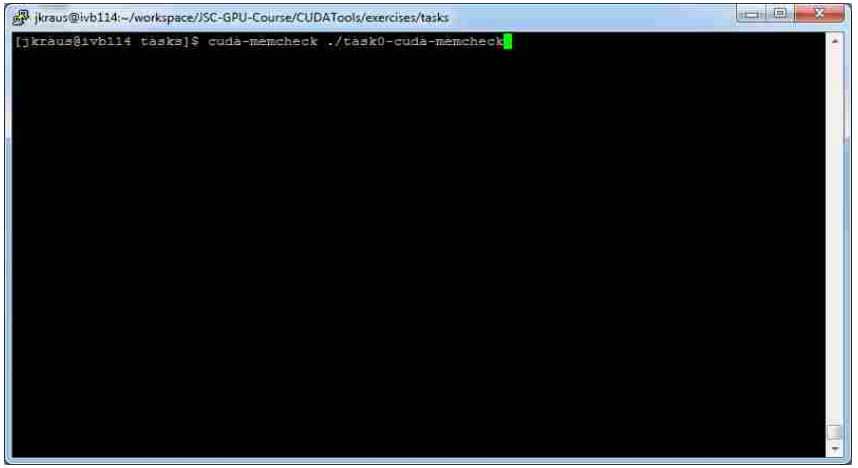
Score-P

- CUDA, OpenCL, OpenACC

CUDA Toolkit

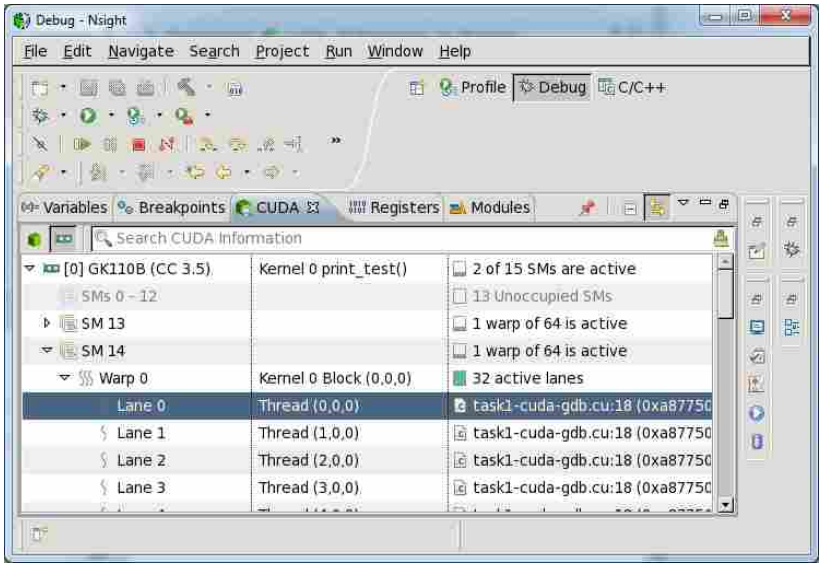
cuda-memcheck:

- memory correctness tool similar to valgrind memcheck
- cuda-memcheck provided to tools (select via –tool)
  - memcheck: Memory access checking
  - racecheck: Shared memory hazard checking
- Compile with debugg information (-g -G)



Nsight Eclipse Edition:

- Source Editor with CUDA C and C++ syntax highlighting
- Project and files management with version control integration
- Integrated build system
- GUI for debugging heterogeneous applications
- Visual profiler integration



CUDA Toolkit

nvprof:

- Command line profiler to get profiles of the application
- Profiles CUDA kernels and API calls

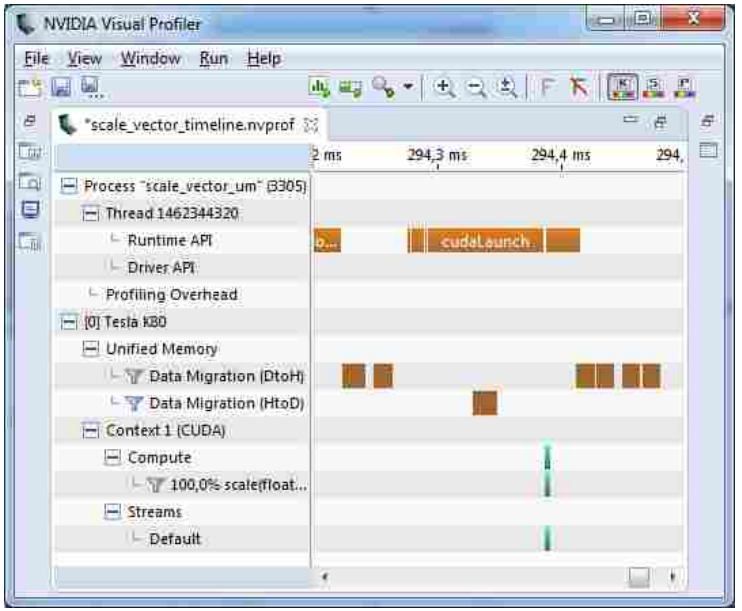
```
> nvprof --unified-memory-profiling per-process-device ./scale_vector_um

==32717== NVPROF is profiling process 32717, command: ./scale_vector_um

==32717== Warning: Unified Memory Profiling is not supported on the current configuration
because a pair of devices without peer-to-peer support is detected on this multi-GPU
setup. When peer mappings are not available, system falls back to using zero-copy memory.
It can cause kernels, which access unified memory, to run slower. More details can be
found at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-managed-
memory

Passed!

==32717== Profiling application: ./scale_vector_um

==32717== Profiling result:

Time(%)      Time     Calls       Avg       Min       Max  Name

100.00%  6.4320us         1  6.4320us  6.4320us  6.4320us  scale(float, float*, float*,
int)

[…] snip
```

NVIDIA Visual Profiler nvvp:

- nvprof can write the application timeline to nvvp compatible file:

```
nvprof --unified-memory-profiling per-process-device
-o scale_vector.nvprof ./scale_vector_um
```

- import in nvvp



GPU Programming Guidance and Support at JSC

- JURECA user info: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html
  - GPU Computing: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/UserInfo/GpuNodes.html?nn=1803700
- Contact user support: sc@fz-juelich.de
- NVIDIA Application Lab at Jülich:
  - jointly operated by Jülich Supercomputing Centre (JSC) and NVIDIA
    - enable scientific applications for GPU-based architectures
    - provide support for their optimization
    - investigate performance and scaling
    - contact: d.pleiter@fz-juelich.de
- JSC many-core interest group:
  - JSC GPU programming courses (CUDA, OpenACC, OpenCL)
  - bi-weekly meeting, contact: w.homberg@fz-juelich.de