Using JURECA's GPU Nodes

Willi Homberg

Introduction to the usage and programming of supercomputer resources in Jülich
23-24 May 2016
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

© NVIDIA Corporation 2010
GPU Computing

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!

Fig.: Nvidia

- 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

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
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

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

```c
__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 GPU memory
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


### TECHNICAL SPECIFICATIONS

<table>
<thead>
<tr>
<th></th>
<th>Tesla K40</th>
<th>Tesla K80¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak double-precision floating point performance (board)</td>
<td>1.43 Tflops</td>
<td>1.87 Tflops</td>
</tr>
<tr>
<td>Peak single-precision floating point performance (board)</td>
<td>4.29 Tflops</td>
<td>5.6 Tflops</td>
</tr>
<tr>
<td>GPU</td>
<td>1 x GK110B</td>
<td>2 x GK210</td>
</tr>
<tr>
<td>CUDA cores</td>
<td>2,880</td>
<td>4,992</td>
</tr>
<tr>
<td>Memory size per board (GDDR5)</td>
<td>12 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Memory bandwidth for board (ECC off)²</td>
<td>288 Gbytes/sec</td>
<td>480 Gbytes/sec</td>
</tr>
<tr>
<td>Architecture features</td>
<td>SMX, Dynamic Parallelism, Hyper-Q</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>Servers and workstations</td>
<td>Servers</td>
</tr>
</tbody>
</table>

¹: The Tesla K80 board contains two Tesla K210 GPUs.
²: Memory bandwidth for ECC (Error Check and Correction) disabled.
GPU APIs and Libraries

APIs

- CUDA
  - Programming model developed by NVIDIA which only supports NVIDIA 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

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

![DGEMM Benchmark](image)
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. 86/2016 in the training programme of Forschungszentrum Jülich)

- course announcement:

- 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

GPU Programming with OpenACC

Pragma/directive based programming 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. 87/2016 in the training programme of Forschungszentrum Jülich)

- course announcement:

- 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

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. 88/2016 in the training programme of Forschungszentrum Jülich)
  
  - course announcement:
  
  - 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

Unified Memory: Access to Host and Device Data

Traditional Developer View

```
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);
}
```

Developer View With Unified Memory

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

    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);

    qsort<<<...>>>(data, N, 1, compare);
    cudaMemcpy(data_d, data, N,
                cudaMemcpyDeviceToHost);
    use_data(data);
    cudaFree(data_d); free(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 is 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;
}
```

`srun -n 4 ./myapp <args>`
**MPI + CUDA**

---

**With UVA and CUDA-aware MPI**

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

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

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

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

---

**No UVA and regular MPI**

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

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

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

```c
//MPI rank n-1
MPI_Recv(r_buf_h, size, ...);
cudaMemcpy(r_buf_d, r_buf_h, size, ...);
```
Compile and Build Executable

CUDA:

- **GCC**
  - # not the standard GCC compiler module
  - module load GCC/4.9.3-2.25
- **wrapper nvcc_icpc**
  - module load Intel/2015.3.187-GCC-4.9.3-2.25

module load CUDA

nvcc -o prog prog.cu

cuBLAS:

module load Intel/2015.3.187-GCC-4.9.3-2.25

module load MVAPICH2

module load CUDA

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

module load GCC/4.9.3-2.25

INC=-I${CUDA_HOME}/include

LIB=-L${CUDA_HOME}/lib64

gcc -o prog prog.c -IOpenCL $(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

```bash
#!/bin/bash -xe
#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

```bash
#!/bin/bash -x
#SBATCH --nodes=1
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:00:00
#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
- soon: CUDA, OpenCL, OpenACC (under development)
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

NVIDIA Visual Profiler nvvp:
- nvprof can write the application timeline to nvvp compatible file:
  ```
  nvprof --unified-memory-profiling per-process-device ./scale_vector_um
  -o scale_vector.nvprof ./scale_vector_um
  ```
- import in nvvp

```bash
> 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

Passed!

--32717== Profiling application: ./scale_vector_um

--32717== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00%</td>
<td>6.4320us</td>
<td>1</td>
<td>6.4320us</td>
<td>6.4320us</td>
<td>6.4320us</td>
<td>scale(float, float*, float*, int)</td>
</tr>
</tbody>
</table>
```

[Image of NVIDIA Visual Profiler]
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](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html)
- 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
Extra Slides
Getting a Feeling for GPU Performance on JURECA

JURECA single node:
1 GPU vs.
2 CPUs, 24 cores
Parallel Computing (I)

Amdahl’s Law

Runtime on single processor:

\[ T_{\text{total}}(1) = T_{\text{setup}} + T_{\text{compute}} + T_{\text{finalization}} \]

Runtime on \( P \) processors:

\[ T_{\text{total}}(P) = T_{\text{setup}} + \frac{T_{\text{compute}}(1)}{P} + T_{\text{finalization}} \]

Speedup:

\[ S(P) = \frac{T_{\text{total}}(1)}{T_{\text{total}}(P)} \]

Serial fraction \( \gamma \):

\[ \gamma = \frac{T_{\text{setup}} + T_{\text{finalization}}}{T_{\text{total}}(1)} \]

Runtime on \( P \) processors:

\[ T_{\text{total}}(P) = \gamma T_{\text{total}}(1) + \frac{(1 - \gamma) T_{\text{total}}(1)}{P} \]

Amdahl’s law:

\[ S(P) = \frac{T_{\text{total}}(1)}{\gamma T_{\text{total}}(1) + \frac{(1 - \gamma) T_{\text{total}}(1)}{P}} = \frac{1}{\gamma + \frac{1 - \gamma}{P}} \]
Using highly parallel computers (and accelerators like GPUs) only makes sense for programs with minimal serial code fractions!
Parallel Computing (III)
Gustafson-Barsis's Law

... speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size.


Amdahl's Law: problem size fix, minimise time-to-solution
Gustafson's Law: execution time fix, increase # processors
Serial fraction $\gamma$

Runtime on $P$ processors in parallel: $\gamma + (1 - \gamma) = 1$

Runtime on $P$ processors (hypothetical serial): $\gamma + P (1 - \gamma)$

Speedup: $S(P) = \gamma + P (1 - \gamma) = P - \gamma (P - 1)$
→ a sufficient large problem can be parallelised efficiently
Parallel Computing (IV)

Gustafson-Barsis's Law

http://en.wikipedia.org/wiki/Gustafson’s_law

a sufficient large problem can be parallelised efficiently
Matrix Multiply $C = A \times B$

Matrix sizes:

Size of $A$: $m_A \times n_A$

(# of rows * # of columns)

(height * width)

Size of $B$: $m_B \times n_B$

Size of $C$: $m_C \times n_C$

Precondition: $n_A = m_B$

In result:

$m_C = m_A$,

$n_C = n_B$

Formula:

$$c_{i,j} = \sum_{e=0}^{n_A-1} a_{i,e} b_{e,j}$$
2D Arrays: Address/Index Computation
(row-major, with stride)

Formulas:

\[ \text{offset} = y \times \text{stride} + x \]
\[ x = \text{offset} \mod \text{stride} \]
\[ y = \text{offset} / \text{stride} \]

(using integer arithmetics, indices start with 0)
Simple Matrix Multiply: Host Code

Simple C struct to hold matrix data:

```c
// basic matrix data type
struct simpleMatrix {
    unsigned int width;
    unsigned int height;
    unsigned int stride;
    size_t bufferSize;
    real_t* ptr;
};
```

Matrix multiplication on the host:

```c
// no-frills CPU implementation of matrix multiplication
void multCPU( simpleMatrix &C, const simpleMatrix &A,
              const simpleMatrix &B )
{
    for( int y=0; y < C.height; y++ ) {
        for( int x=0; x < C.width; x++ ) {
            real_t CVal = 0.0;
            for( int e=0; e < A.width; e++ ) {
                CVal += A.ptr[y*A.stride+e] * B.ptr[e*B.stride+x];
            }
            C.ptr[y*C.stride+x] = CVal;
        }
    }
    return;
}
```
Simple Matrix Multiply: OpenCL Kernel Code

```c
__kernel _attribute__((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute__((vec_type_hint(real_t)))

__kernel void matMulKernel(
    int Aheight, int Awidth,
    int Bheight, int Bwidth,
    int Astride, int Bstride, int Cstride,
    __global real_t* Aelements,
    __global real_t* Belements,
    __global real_t* Celements)
{
    // Get global indices of work-item
    int global_row = get_global_id(1);
    int global_col = get_global_id(0);

    // Check if we are within valid area of matrix C
    if( global_row < Aheight && global_col < Bwidth ) {
        // Compute single element of C
        real_t Cvalue = 0;
        for (int e = 0; e < Awidth; ++e)
            Cvalue += Aelements[global_row * Astride + e] * Belements[e * Bstride + global_col];
        // Write result into C matrix
        Celements[global_row * Cstride + global_col] = Cvalue;
    }
}
```
Runtime Comparison between Devices (Simple Matrix Multiplication)

Comparison between Devices [DP] [SQUARE] [comp]

- **AMD S10000** (GPU) [DP, non-tiled (B=16)]
  \[ R_{\text{peak}, \text{DP}} \approx 730 \text{ GFlops} \]

- **Xeon Phi** (MIC) [DP, non-tiled (B=16)]
  \[ R_{\text{peak}, \text{DP}} \approx 1000 \text{ GFlops} \]

- **Xeon E5-2650 (2x)** [CPU/INTEL-OCL] [DP, non-tiled (B=16)]
  \[ R_{\text{peak}, \text{DP}} \approx 300 \text{ GFlops} \]

- **NVIDIA K40** (GPU) [DP, non-tiled (B=16)]
  \[ R_{\text{peak}, \text{DP}} \approx 1660 \text{ GFlops} \]
Block Matrix Multiply: OpenCL Kernel Code
(Overview)

**Basic idea:** Every work-group is responsible for a different submatrix within C
Block Matrix Multiply:
OpenCL Kernel Code with Local Memory

```c
__kernel void matMulKernel_LM(...) {
  
  for (int m = 0; m < mMx; ++m) {
    // Get base indices of sub-matrices Asub of A and Bsub of B
    Asub_baseIndex = A_stride * BLOCK_SIZE * blockRow + BLOCK_SIZE * m;
    Bsub_baseIndex = B_stride * BLOCK_SIZE * m + BLOCK_SIZE * blockCol;
    
    // Load Asub and Bsub from global memory to local memory
    // Each thread loads one element of each sub-matrix
    Asub_index = Asub_baseIndex + row * A_stride + col;
    if (Asub_index < A_numElems)
      As[row*BLOCK_SIZE+col] = A_elements[Asub_index];
    Bsub_index = Bsub_baseIndex + row * B_stride + col;
    if (Bsub_index < B_numElems)
      Bs[row*BLOCK_SIZE+col] = B_elements[Bsub_index];
    
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    barrier(CLK_LOCAL_MEM_FENCE);
    
    // Multiply row of Asub and column of Bsub together
    // (only iterate up to eMax to prevent inclusion of invalid elements from
    // Asub and Bsub)
    int eMax;
    if (m == (mMx-1) && (r != 0))
      eMax = r;
    else
      eMax = BLOCK_SIZE;
    for (int e = 0; e < eMax; ++e)
      Cvalue += As[row*BLOCK_SIZE+e] * Bs[e*BLOCK_SIZE+col];
    
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  
  ...
}
```
Block Matrix Multiply:
OpenCL Kernel Code with Local Memory

NVIDIA K40 (GPU) [SP] [SQUARE] [comp]
Block Matrix Multiply: Further optimisations
Data Prefetching, Double Buffering, optimum work-group size
nvvp introduction
Cheat Sheet

- Generate timeline with nvprof

  nvprof --unified-memory-profiling per-process-device
  -o <output-profile> ./a.out

- Collect analysis metrics nvprof

  nvprof --unified-memory-profiling per-process-device
  --analysis-metrics -o <output-profile> ./a.out

- Start nvvp

  nvvp

- profiler users guide

Score-P - Profiling / Runtime summarization

- Recording of aggregated information
  - Total, maximum, minimum, ...
- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, ...
  - Processes, threads

Profile = summarization of events over execution interval
Tracing with Score-P and Vampir

- Recording information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events
  - Event trace = Chronologically ordered sequence of event records
- Visualization of dynamics of complex parallel processes:
  - Monitor/Collector (Score-P)
  - Charts/Browser (Vampir)

Typical questions that Vampir helps to answer:
- What happens in my application execution during a given time in a given process or thread?
- How do the communication patterns of my application execute on a real system?
- Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?