Side by side maximization of GPU/MIC accelerators performance by example of numerical simulation problem

D. Mikushin, O. Schenk, A. Ivakhnenko, A. Shevchenko

August 29, 2014
Three types of modern accelerators

**GPU: NVIDIA Tesla K20c**
- Kepler GK110, 28 nm
- 13 mp × 192 cores @ 0.71 GHz
- 5 GB GDDR5 @ 2.6 GHz
- 225W
- ECC: yes

**GPU: Gigabyte Radeon HD 7970**
- Graphics Core Next, 28 nm
- 32 mp × 64 cores @ 1 GHz
- 3GB GDDR5 @ 1.5 GHz
- 250W
- ECC: no

**MIC: Intel Xeon Phi 3120A**
- Knights Corner (KNC), 22 nm
- 57 cores @ 1.1 GHz
- 6GB GDDR5 @ 1.1 GHz
- 300W
- ECC: yes
- up to 4 threads per core
- 512-bit vectorization (AVX-512)
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- **Compare the execution time against CPU version**
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100\times$ speedup and more 😊

- **Compare performance against standard libraries**
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- **Detailed analysis of the target GPU application**
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get 100× speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100\times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    $\Rightarrow$ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
    - Analyze with respect to peak properties of the target GPU
    - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
    - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100\times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- “Parallel STL”: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- “Parallel STL”: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- **“Parallel STL”**: Intel TBB, NVIDIA Thrust, AMD Bolt
- OpenMP/OpenACC: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- **“Parallel STL”**: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- “**Parallel STL**”: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- “**Parallel STL**”: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- “**Parallel STL**”: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem
   (see examples in the next slides)
Compare performance against standard libraries

- **BLAS**: Intel MKL, NVIDIA CUBLAS, AMD clBlas
- "**Parallel STL**: Intel TBB, NVIDIA Thrust, AMD Bolt
- **OpenMP/OpenACC**: Intel, PGI, etc.

Possible uses in analysis:

1. Fast approximation of application performance
2. Compare similar hand-written kernels, when they are necessary
3. Fast choice of the most prospective accelerator for the given problem (see examples in the next slides)
BLAS DGEMM

![Bar chart]

(higher is better)

<table>
<thead>
<tr>
<th>hardware</th>
<th>#cores</th>
<th>software</th>
<th>compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5-2670 × 2</td>
<td>8 × 2</td>
<td>MKL</td>
<td>icpc -g -O3 -xHost</td>
</tr>
<tr>
<td>Xeon 5110P</td>
<td>60</td>
<td>MKL, native</td>
<td>icpc -g -O3 -mmic</td>
</tr>
<tr>
<td>Radeon HD7970</td>
<td>2,048</td>
<td>clAmdBlas</td>
<td>g++ -g -O3</td>
</tr>
<tr>
<td>Kepler K20X</td>
<td>2,496</td>
<td>CUBLAS</td>
<td>g++ -g -O3</td>
</tr>
</tbody>
</table>

icpc/mkl: 2013.2.146, cublas: 5.5
BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 
n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);
cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, 
n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();
cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 
n, n, n, alpha, A, n, B, n, beta, C, n);

gpu@

cblasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, 
n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);
cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

#include <cuda_runtime.h>
#include <cublas_v2.h>

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);  
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);  
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);
cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 
n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, 
n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 
n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);
cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, 
n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();
cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
BLAS DGEMM

CPU BLAS

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
    n, n, n, alpha, A, n, B, n, beta, C, n);
```

GPU-enabled BLAS

```c
#include <cuda_runtime.h>
#include <cublas_v2.h>

double *A_dev = NULL, *B_dev = NULL, *C_dev = NULL;
cudaMalloc(&A_dev, n * n * sizeof(double));
cudaMalloc(&B_dev, n * n * sizeof(double));
cudaMalloc(&C_dev, n * n * sizeof(double));

cublasHandle_t handle;
cublasCreate(&handle);

cublasSetMatrix(n, n, sizeof(double), A, n, A_dev, n);
cublasSetMatrix(n, n, sizeof(double), B, n, B_dev, n);
cublasSetMatrix(n, n, sizeof(double), C, n, C_dev, n);

double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
    n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

cublasGetMatrix(n, n, sizeof(double), C_dev, n, C, n);
cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cublasDestroy(handle);
```
**CPU BLAS**

```c
#include <mkl.h>

double alpha = 1.0, beta = 0.0;
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n, n, n, alpha, A, n, B, n, beta, C, n);
```

**GPU-enabled BLAS**

CUDA 6.0 introduces **NVBLAS** library:

- Automatically routes standard BLAS3 calls to CUBLAS (by preloading libnvblas.so)
- Can spread work across multiple GPUs (cuBLAS-XT)
- Usable with any app that calls BLAS3 functions (Octave, Scilab, etc.)

```c
double alpha = 1.0, beta = 0.0;
dgemm_("n", "n", 
        &n, &n, &n, &alpha, A, &n, B, &n, &beta, C, &n);
```
Sorting 128M key-value pairs

<table>
<thead>
<tr>
<th>hardware</th>
<th>#cores</th>
<th>software</th>
<th>compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5-2670 × 2</td>
<td>8 × 2</td>
<td>TBB, mergesort</td>
<td>icpc -ipo -O3 -no-prec-div -xHost</td>
</tr>
<tr>
<td>Xeon 5110P</td>
<td>60</td>
<td>TBB, mergesort, native</td>
<td>icpc -ipo -O3 -no-prec-div -mmic</td>
</tr>
<tr>
<td>Radeon HD79790</td>
<td>2,048</td>
<td>Bolt, GPU sort</td>
<td>g++ -g -O3 -std=c++0x</td>
</tr>
<tr>
<td>Kepler K20X</td>
<td>2,496</td>
<td>Thrust, GPU radix sort</td>
<td>nvcc -arch=sm_35 -O3</td>
</tr>
</tbody>
</table>

ICPC: 2013.2.146, tbb: 4.1, bolt: 1.1, thrust: 5.5
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

// Generate the random keys and values on the host.
host_vector<float> h_keys(n);
host_vector<float> h_vals(n);
for (int i = 0; i < n; i++)
{
    h_keys[i] = drand48();
    h_vals[i] = drand48();
}

// Transfer data to the device.
device_vector<float> d_keys = h_keys;
device_vector<float> d_vals = h_vals;

// Sort!
sort_by_key(d_keys.begin(), d_keys.end(), d_vals.begin());
cudaDeviceSynchronize();

// Transfer data back to host.
copy(d_keys.begin(), d_keys.end(), h_keys.begin());
copy(d_vals.begin(), d_vals.end(), h_vals.begin());

Relatively new CUB library from NVResearch offers radix sort with 2.5 × higher perf than Thrust!
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

// Generate the random keys and values on the host.
host_vector<float> h_keys(n);
host_vector<float> h_vals(n);
for (int i = 0; i < n; i++)
    h_keys[i] = drand48();
    h_vals[i] = drand48();

// Transfer data to the device.
device_vector<float> d_keys = h_keys;
device_vector<float> d_vals = h_vals;

// Sort!
sort_by_key(d_keys.begin(), d_keys.end(), d_vals.begin());
cudaDeviceSynchronize();

// Transfer data back to host.
copy(d_keys.begin(), d_keys.end(), h_keys.begin());
copy(d_vals.begin(), d_vals.end(), h_vals.begin());
Wave propagation stencil

(higher is better)

<table>
<thead>
<tr>
<th>hardware</th>
<th>#cores</th>
<th>software</th>
<th>compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5-2670 × 2</td>
<td>8 × 2</td>
<td>C, OpenMP</td>
<td>icc -O3 -xHost -openmp -opt-subscript-in-range -align</td>
</tr>
<tr>
<td>Xeon 5110P</td>
<td>60</td>
<td>C, OpenMP, offload</td>
<td>icc -O3 -xHost -openmp -opt-subscript-in-range -align</td>
</tr>
<tr>
<td>Radeon HD7970</td>
<td>2,048</td>
<td>OpenACC</td>
<td>pgcc -acc -O3 -ta=radeon,tahiti</td>
</tr>
<tr>
<td>Kepler K20X</td>
<td>2,496</td>
<td>OpenACC</td>
<td>pgcc -acc -O3 -ta=nvidia:keep,cc35,cuda5.5</td>
</tr>
</tbody>
</table>

icpc: 2013.2.146, pgi: 14.1 – nvidia, 14.6 – radeon
```c
void wave13pt(const int nx, const int ny, const int ns,
               const real m0, const real m1, const real m2,
               const real* const __restrict__ w0, const real* const __restrict__ w1,
               real* const __restrict__ w2)
{
    size_t szarray = (size_t)nx * ny * ns;
    #pragma acc kernels loop independent gang(ns), present(w0[0:szarray], w1[0:szarray], w2[0:szarray])
    for (int k = 2; k < ns - 2; k++)
    {
        #pragma acc loop independent
        for (int j = 2; j < ny - 2; j++)
        {
            #pragma acc loop independent vector(512)
            for (int i = 2; i < nx - 2; i++)
            {
                _A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
                m1 * (_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
                       _A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
                       _A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
                m2 * (_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
                       _A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
                       _A(w1, k+2, j, i) + _A(w1, k-2, j, i));
            } // i-loop
        } // j-loop
    } // k-loop
}
```
void wave13pt(const int nx, const int ny, const int ns, 
const real m0, const real m1, const real m2, 
const real* const __restrict__ w0, const real* const __restrict__ w1,  
real* const __restrict__ w2)
{
    size_t szarray = (size_t)nx * ny * ns;
    #pragma acc kernels loop independent gang(ns), present(w0[0:szarray], w1[0:szarray], w2[0:szarray])
    for (int k = 2; k < ns − 2; k++)
    {
        #pragma acc loop independent
        for (int j = 2; j < ny − 2; j++)
        {
            #pragma acc loop independent vector(512)
            for (int i = 2; i < nx − 2; i++)
            {
                _A(w2, k, j, i) = m0 * _A(w1, k, j, i) − _A(w0, k, j, i) + 

                m1 * (  
                    _A(w1, k, j, i+1) + _A(w1, k, j, i−1) +  
                    _A(w1, k, j+1, i) + _A(w1, k, j−1, i) +  
                    _A(w1, k+1, j, i) + _A(w1, k−1, j, i)) +  

                m2 * (  
                    _A(w1, k, j, i+2) + _A(w1, k, j, i−2) +  
                    _A(w1, k, j+2, i) + _A(w1, k, j−2, i) +  
                    _A(w1, k+2, j, i) + _A(w1, k−2, j, i));
            } // i-loop
        } // j-loop
    } // k-loop
}
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    $\Rightarrow$ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- Detailed analysis of the target GPU application
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Compute-bound VS memory-bound

- **Compute-bound** – application heavily utilizes compute units, memory dataflow is below maximum
- **Memory-bound** – application intensively reads/writes global memory (DRAM, not cache), while compute units are stalled waiting for data in memory
- **Balanced** – compute units and memory are both close to their maximum utilization (ideal case)

Optimizations can turn program from compute-bound into memory-bound and vise versa!
Compute-bound VS memory-bound

- **Compute-bound** – application heavily utilizes compute units, memory dataflow is below maximum
- **Memory-bound** – application intensively reads/writes global memory (DRAM, not cache), while compute units are stalled waiting for data in memory
- **Balanced** – compute units and memory are both close to their maximum utilization (ideal case)

![Graph showing the relationship between arithmetic intensity and attainable GFlops/s.](image)

- Optimizations can turn program from compute-bound into memory-bound and vise versa!
- **Compute-bound** – application heavily utilizes compute units, memory dataflow is below maximum
- **Memory-bound** – application intensively reads/writes global memory (DRAM, not cache), while compute units are stalled waiting for data in memory
- **Balanced** – compute units and memory are both close to their maximum utilization (ideal case)

![Memory bound vs Compute bound](image)

- Optimizations can turn program from compute-bound into memory-bound and vise versa!
- **Compute-bound** – application heavily utilizes compute units, memory dataflow is below maximum

- **Memory-bound** – application intensively reads/writes global memory (DRAM, not cache), while compute units are stalled waiting for data in memory

- **Balanced** – compute units and memory are both close to their maximum utilization (ideal case)

- Optimizations can turn program from compute-bound into memory-bound and vise versa!
Compute-bound vs memory-bound

- **Compute-bound** – application heavily utilizes compute units, memory dataflow is below maximum
- **Memory-bound** – application intensively reads/writes global memory (DRAM, not cache), while compute units are stalled waiting for data in memory
- **Balanced** – compute units and memory are both close to their maximum utilization (ideal case)

- Optimizations can turn program from compute-bound into memory-bound and vice versa!
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
  - ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- **Detailed analysis of the target GPU application**
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
  - Profiling
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:
  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (Q: examples for GPU?)
  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (Q: examples for GPU?)
  - If memory-bound application cannot utilize the whole memory throughput, then it's possibly suboptimal (Q: examples for GPU?)
Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:

- If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (Q: examples for GPU?)

- If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (Q: examples for GPU?)

- If memory-bound application cannot utilize the whole memory throughput, then it's possibly suboptimal (Q: examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:
  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (Q: examples for GPU?)
  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (Q: examples for GPU?)
  - If memory-bound application cannot utilize the whole memory throughput, then it's possibly suboptimal (Q: examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:
  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (Q: examples for GPU?)
    Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)
  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (Q: examples for GPU?)
  - If memory-bound application cannot utilize the whole memory throughput, then it’s possibly suboptimal (Q: examples for GPU?)
Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:

- If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (Q: examples for GPU?)
  
  Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)

- If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (Q: examples for GPU?)

- If memory-bound application cannot utilize the whole memory throughput, then it’s possibly suboptimal (Q: examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:

  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (*Q:* examples for GPU?)
    
    Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)

  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (*Q:* examples for GPU?)
    
    Caching (in registers, in shared memory *GPU/GPU*)

  - If memory-bound application cannot utilize the whole memory throughput, then it’s possibly suboptimal (*Q:* examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:

  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (*Q:* examples for GPU?)
    
    Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)

  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (*Q:* examples for GPU?)
    
    Caching (in registers, in shared memory **GPU/GPU**)  
    “On-the-fly” computations

  - If memory-bound application cannot utilize the whole memory throughput, then it's possibly suboptimal (*Q:* examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:

  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (**Q**: examples for GPU?)
    
    Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)
  
  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (**Q**: examples for GPU?)
    
    Caching (in registers, in shared memory **GPU/GPU**)
    
    “On-the-fly” computations

  - If memory-bound application cannot utilize the whole memory throughput, then it’s possibly suboptimal (**Q**: examples for GPU?)
Global memory throughput analysis

- Hardware manufacturer can write *any* memory bandwidth in specs – only that value makes sense, which is practically reachable in simple tests:
  
  - If memory-bound application produces high memory throughput, it does not yet mean the memory throughput is utilized efficiently (**Q:** examples for GPU?)
    
    Bad coalescing (one thread requests 8 bytes – reads 128 and throws away 120, if neighbouring threads don’t need them)
  
  - If memory-bound application already utilizes memory throughput efficiently, then optimizations aiming to reduce memory pressure potentially make most of sense (**Q:** examples for GPU?)
    
    Caching (in registers, in shared memory GPU/GPU)
    “On-the-fly” computations
  
  - If memory-bound application cannot utilize the whole memory throughput, then it’s possibly suboptimal (**Q:** examples for GPU?)
    
    Insufficient parallelism (not enough blocks GPU/GPU) and/or threads GPU/GPU/MIC
Global memory throughput analysis

Transferred data buffer size, bytes

Memory throughput, Gb/sec

(higher is better)

<table>
<thead>
<tr>
<th>hardware</th>
<th>#cores</th>
<th>benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon 5110P</td>
<td>60</td>
<td>STREAM benchmark</td>
</tr>
<tr>
<td>Xeon 3120A</td>
<td>57</td>
<td>STREAM benchmark</td>
</tr>
<tr>
<td>Radeon HD7970</td>
<td>2,048</td>
<td>APPSDK</td>
</tr>
<tr>
<td>Kepler K20X</td>
<td>2,496</td>
<td>CUDA SDK / bandwidthTest</td>
</tr>
</tbody>
</table>
Global memory throughput analysis: differences between generations of MIC

MIC has 8 memory controllers in total, with 2 32-bit channels each

- On 3120 card 4 channels are disabled, only 12 are operational
- 3120: 32 bits $\times$ 12 channels $\times$ 2,5GHz $\times$ 2-way transfer = 240 Gb/sec
- 5110: 32 bits $\times$ 16 channels $\times$ 2,5GHz $\times$ 2-way transfer = 320 Gb/sec

(higher is better)
Non-linear equation of diffusion-reaction (Fisher equation):

$$\frac{\delta^2 u}{\delta t^2} = D \frac{\delta^2 u}{\delta x^2} + Ru(1 - u)$$

- Rectangular problem domain
- Dirichlet boundary conditions

Source code for CPU, MIC, GPU: https://github.com/apc-llc/summer-school
1: \textbf{procedure} mini-stencil
2: \hspace{1em} x_{old} \leftarrow 0
3: \hspace{1em} bnd_N \leftarrow 0; bnd_S \leftarrow 0
4: \hspace{1em} bnd_E \leftarrow 0; bnd_W \leftarrow 0
5: \hspace{1em} \Delta x \leftarrow 0
6: \hspace{1em} \text{tolerance} \leftarrow 10^{-6}
7: \text{time steps iterations:}
8: \hspace{1em} \textbf{for} i = 1, i_{\text{max}} \textbf{do}
9: \hspace{2em} x_{old} \leftarrow x_{\text{new}}
10: \hspace{1em} \text{non-linear iterations:}
11: \hspace{2em} \textbf{for} j = 1, j_{\text{max}} \textbf{do}
12: \hspace{3em} \text{diffusion}(x_{\text{new}}, x_{old}, b);
13: \hspace{3em} \textbf{if} \ |b| < \text{tolerance} \textbf{then break}
14: \hspace{3em} is_{\text{converged}} \leftarrow \text{cg}(N, \delta x, b, cg_{\text{max}}, \text{tolerance})
15: \hspace{3em} \textbf{if} \ \text{not} \ is_{\text{converged}} \textbf{then error}
16: \hspace{3em} x_{\text{new}} \leftarrow x_{\text{new}} - \delta x;
procedure mini-stencil

2. \( x_{\text{old}} \leftarrow 0 \)

3. \( b_{\text{N}} \leftarrow 0; b_{\text{S}} \leftarrow 0 \)

4. \( b_{\text{E}} \leftarrow 0; b_{\text{W}} \leftarrow 0 \)

5. \( \Delta x \leftarrow 0 \)

6. \( \text{tolerance} \leftarrow 10^{-6} \)

7. time steps iterations:

8. \( \text{for } i = 1, i_{\text{max}} \) do

9. \( x_{\text{old}} \leftarrow x_{\text{new}} \)

10. non-linear iterations:

11. \( \text{for } j = 1, j_{\text{max}} \) do

12. diffusion\((x_{\text{new}}, x_{\text{old}}, b)\);

13. \( \text{if } ||b|| < \text{tolerance} \) then break

14. is_converged \( \leftarrow \) cg\((N, \delta x, b, cg_{\text{max}}, \text{tolerance})\)

15. \( \text{if not is_converged} \) then error

16. \( x_{\text{new}} \leftarrow x_{\text{new}} - \delta x; \)

- “Outer” time step iterations – \( i \)
- “Inner” non-linearity iterations – \( j \)
- Diffusion operator
- Solve system of linear equations with conjugate gradients method
1: procedure mini-stencil
2: \( x_{\text{old}} \leftarrow 0 \)
3: \( bnd_N \leftarrow 0; bnd_S \leftarrow 0 \)
4: \( bnd_E \leftarrow 0; bnd_W \leftarrow 0 \)
5: \( \Delta x \leftarrow 0 \)
6: tolerance \( \leftarrow 10^{-6} \)
7: time steps iterations:
8: \( \text{for } i = 1, i_{\text{max}} \text{ do} \)
9: \( x_{\text{old}} \leftarrow x_{\text{new}} \)
10: non-linear iterations:
11: \( \text{for } j = 1, j_{\text{max}} \text{ do} \)
12: \( \text{diffusion}(x_{\text{new}}, x_{\text{old}}, b); \)
13: \( \text{if } \| b \| < \text{tolerance} \text{ then break} \)
14: \( \text{is\_converged} \leftarrow \text{cg}(N, \delta x, b, cg_{\text{max}}, \text{tolerance}) \)
15: \( \text{if not is\_converged} \text{ then error} \)
16: \( x_{\text{new}} \leftarrow x_{\text{new}} - \delta x; \)

- “Outer” time step iterations – \( i \)
- “Inner” non-linearity iterations – \( j \)
- Diffusion operator
- Solve system of linear equations with conjugate gradients method
Application No.1: Algorithm

1: procedure mini-stencil
2: \[x_{old} \leftarrow 0\]
3: \[bnd_N \leftarrow 0; bnd_S \leftarrow 0\]
4: \[bnd_E \leftarrow 0; bnd_W \leftarrow 0\]
5: \[\Delta x \leftarrow 0\]
6: \[\text{tolerance} \leftarrow 10^{-6}\]
7: \[\text{time steps iterations:}\]
8: \[\text{for } i = 1, i_{max} \text{ do}\]
9: \[x_{old} \leftarrow x_{new}\]
10: \[\text{non-linear iterations:}\]
11: \[\text{for } j = 1, j_{max} \text{ do}\]
12: \[\text{diffusion}(x_{new}, x_{old}, b);\]
13: \[\text{if } ||b|| < \text{tolerance} \text{ then break}\]
14: \[\text{is_converged} \leftarrow \text{cg}(N, \delta x, b, cg_{max}, \text{tolerance})\]
15: \[\text{if not is_converged then error}\]
16: \[x_{new} \leftarrow x_{new} - \delta x;\]
1: procedure mini-stencil
2: \( x_{\text{old}} \leftarrow 0 \)
3: \( b_{\text{old}}^N \leftarrow 0; b_{\text{old}}^S \leftarrow 0 \)
4: \( b_{\text{old}}^E \leftarrow 0; b_{\text{old}}^W \leftarrow 0 \)
5: \( \Delta x \leftarrow 0 \)
6: \( \text{tolerance} \leftarrow 10^{-6} \)
7: time steps iterations:
8: \( \text{for } i = 1, i_{\text{max}} \text{ do} \)
9: \( x_{\text{old}} \leftarrow x_{\text{new}} \)
10: non-linear iterations:
11: \( \text{for } j = 1, j_{\text{max}} \text{ do} \)
12: \( \text{diffusion}(x_{\text{new}}, x_{\text{old}}, b); \)
13: \( \text{if } ||b|| < \text{tolerance} \text{ then break} \)
14: \( \text{is\_converged} \leftarrow \text{cg}(N, \delta x, b, \text{cg\_max}, \text{tolerance}) \)
15: \( \text{if not is\_converged \ then error} \)
16: \( x_{\text{new}} \leftarrow x_{\text{new}} - \delta x; \)

- “Outer” time step iterations – \( i \)
- “Inner” non-linearity iterations – \( j \)
- Diffusion operator
- Solve system of linear equations with conjugate gradients method
Application No. 1: Implementations

**GPU: NVIDIA Tesla K20c**
- Diffusion kernel in CUDA
- Linear algebra in CUDA
- Reducing synchronization overheads between CUDA kernels in CUDA kernels
- Vectorization of loads and stores
- Kernel specialization
- Dynamic parallelism: all GPU code as a single kernel

**GPU: Gigabyte Radeon HD 7970**
- Diffusion kernel in OpenCL
- Ddot & Daxpy – in clAmdBlas, all the rest – in OpenCL
- Reducing synchronization overheads between OpenCL kernels
- Kernel specialization

**MIC: Intel Xeon Phi 3120A**
- Diffusion and linear algebra in OpenMP
- Manual vectorization with AVX-512
- Loops unrolling
- Reducing synchronization overheads between OpenMP loops
- Use of huge memory pages
Application No.1: Performance results

The table below shows the performance results for various problem domain sizes on different devices:

<table>
<thead>
<tr>
<th>size</th>
<th>Xeon Phi 3110A</th>
<th>Xeon Phi 5110P</th>
<th>Radeon HD7970</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>512^2</td>
<td>10.714055</td>
<td>10.588785</td>
<td>11.189923</td>
<td>13.302244</td>
</tr>
<tr>
<td>1024^2</td>
<td>14.861325</td>
<td>16.489451</td>
<td>17.483508</td>
<td>18.846611</td>
</tr>
<tr>
<td>4096^2</td>
<td>13.306116</td>
<td>18.133706</td>
<td>28.035458</td>
<td>23.144442</td>
</tr>
</tbody>
</table>

(higher is better)
Three methods of analyzing application efficiency on accelerators

- Compare the execution time against CPU version
  - The quality of CPU version optimization is often neglected
    ⇒ good method to get $100 \times$ speedup and more 😊

- Compare performance against standard libraries
  - Ensure the known good performance levels are achieved for “typical” problems
    (e.g. 300 GFLOPS for DP DGEMM on Tesla C2075)

- **Detailed analysis of the target GPU application**
  - Use normalized metrics: FLOPS, FLOP/byte (roofline)
  - Analyze with respect to peak properties of the target GPU
    - Profiling
Intel VTune Amplifier

- Allows fast CPU application profiling, e.g. for preparing accelerator port
- Handy predefined analysis scenarios
- Also profiles Xeon Phi (requires installing of profiling kernel module on Xeon Phi)
- “CPI Rate” metric from “Advanced Hostspots” scenario is similar to FLOPS/byte
- Shows multithreading utilization efficiency
- Free for non-commercial personal use on Linux
### Profiling: MIC counters in VTune

Profiling data from a VTune Amplifier project showing hardware event counts for different functions and source files. The table displays total and self-instructions executed, along with CPU clock unhalted times for various functions.

#### Source Function Stack

<table>
<thead>
<tr>
<th>Source Function Stack</th>
<th>INSTRUCTIONS_EXECUTED</th>
<th>CPU_CLK_UNHALTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>10,040,000,000</td>
<td>54,410,000,000</td>
</tr>
<tr>
<td>libomp5.so</td>
<td>7,030,000,000</td>
<td>54,050,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@140</td>
<td>480,000,000</td>
<td>1,930,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@97</td>
<td>420,000,000</td>
<td>1,800,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@97</td>
<td>380,000,000</td>
<td>1,810,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@119</td>
<td>370,000,000</td>
<td>1,420,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@58</td>
<td>350,000,000</td>
<td>1,120,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@140</td>
<td>340,000,000</td>
<td>1,390,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@58</td>
<td>260,000,000</td>
<td>2,090,000,000</td>
</tr>
<tr>
<td>operators::diffusionSmpParallel@64</td>
<td>100,000,000</td>
<td>1,320,000,000</td>
</tr>
<tr>
<td>main</td>
<td>90,000,000</td>
<td>1,250,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@149</td>
<td>30,000,000</td>
<td>190,000,000</td>
</tr>
<tr>
<td>linalg::ss axpySmpParallel_for@97</td>
<td>30,000,000</td>
<td>90,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@58</td>
<td>60,000,000</td>
<td>260,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@129</td>
<td>20,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@108</td>
<td>20,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@129</td>
<td>20,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>linalg::ss cgSmpParallel_for@83</td>
<td>10,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>[import thunk std::ios_base::Init::Init]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>linalg::ss cg</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>func@0x4018f3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>linalg::ss norm2SomParallel_for@71</td>
<td>0</td>
<td>70,000,000</td>
</tr>
</tbody>
</table>

Selected 1 row(s):

- INSTRUCTIONS_EXECUTED: 10,040,000,000
- CPU_CLK_UNHALTED: 54,410,000,000

**Diagram:**
- Thread
- Running:
- Hardware Event Count

Dmitry Mikushin et al. (Applied Parallel Computing LLC)
Profiling: MIC counters in VTune

### Hardware Event Counts viewpoint (change)

<table>
<thead>
<tr>
<th>Grouping</th>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Function Stack

<table>
<thead>
<tr>
<th>Function Stack</th>
<th>Hardware Event Count: Total by Hardware Event Type</th>
<th>Hardware Event Count: Self by Hardware Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSTRUCTIONS_EXECUTED</td>
<td>CPU_CLK_UNHALTED</td>
</tr>
<tr>
<td>Total</td>
<td>6,060,000,000</td>
<td>30,630,000,000</td>
</tr>
<tr>
<td>[libiomp5.so]</td>
<td>3,130,000,000</td>
<td>19,000,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>570,000,000</td>
<td>1,730,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>530,000,000</td>
<td>1,820,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>500,000,000</td>
<td>1,770,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>470,000,000</td>
<td>1,530,000,000</td>
</tr>
<tr>
<td>operators::diffusion$omp$vec</td>
<td>230,000,000</td>
<td>2,480,000,000</td>
</tr>
<tr>
<td>vmlinux</td>
<td>40,000,000</td>
<td>160,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>2,000,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>20,000,000</td>
<td>70,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>10,000,000</td>
<td>110,000,000</td>
</tr>
<tr>
<td>linalg::ss_csgomp$parallel_for</td>
<td>10,000,000</td>
<td>60,000,000</td>
</tr>
<tr>
<td>main</td>
<td>20,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>_stl_SE</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Selected 1 row(s):**

<table>
<thead>
<tr>
<th>Function Stack</th>
<th>Hardware Event Count: Total by Hardware Event Type</th>
<th>Hardware Event Count: Self by Hardware Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSTRUCTIONS_EXECUTED</td>
<td>CPU_CLK_UNHALTED</td>
</tr>
<tr>
<td>main (TID: 50)</td>
<td>570,000,000</td>
<td>1,730,000,000</td>
</tr>
</tbody>
</table>

---

Dmitry Mikushin et al.  (Applied Parallel Computing LLC)  
GPU/APU/MIC optimization  
August 29, 2014  
24 / 50
NVIDIA Visual Profiler

- Graphical profiler based on Eclipse
- Profiles on local GPU, starting from CUDA 6.0 can also profile on remote GPU, regardless local/remote OS (which is very very handy!)
- Supports user-defined profiling regions

Issues:

- Based on Eclipse ⇒ starts to lag with anything more complex than a simple test
- While collecting profile, application usually gets executed multiple times
- For large applications the time to wait for profiling results → ∞
Application No.1: Implementations

**GPU:** NVIDIA Tesla K20c
- Diffusion kernel in CUDA
- Linear algebra in CUDA
- Reducing synchronization overheads between CUDA kernels in CUDA kernels
- **Vectorization of loads and stores**
- Kernel specialization
- Dynamic parallelism: all GPU code as a single kernel

**GPU:** Gigabyte Radeon HD 7970
- Diffusion kernel in OpenCL
- Ddot & Daxpy – in clAmdBlas, all the rest – in OpenCL
- Reducing synchronization overheads between OpenCL kernels
- Kernel specialization

**MIC:** Intel Xeon Phi 3120A
- Diffusion and linear algebra in OpenMP
- **Manual vectorization with AVX-512**
- Loops unrolling
- Reducing synchronization overheads between OpenMP loops
- Use of huge memory pages
All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.

2012’ and newer CPU can process 4 doubles with one instruction (AVX).

Running code without vectorization \(\Rightarrow\) use of only \(\approx1/4\) of CPU’s core compute power.

Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.

\(\Rightarrow\) we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

```c
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void*) (y[8*i]), _mm512_add_pd(_mm512_load_pd((void*) (y[8*i])),
                 _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((void*) (x[8*i]))));
```

Dmitry Mikushin et al. (Applied Parallel Computing LLC)
All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.

- 2012’ and newer CPU can process 4 doubles with one instruction (AVX).
- Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power.
- Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.
- ⇒ we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

```c
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((__m512 *) (y + 8 * i),
    _mm512_add_pd(_mm512_load_pd((__m512 *) (y + 8 * i)),
    _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((__m512 *) (x + 8 * i))));
```
Vectorization on modern CPUs and MIC: AVX

- All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously
- 2012’ and newer CPU can process 4 doubles with one instruction (AVX)
- Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power
- Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only
- ⇒ we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

```c
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void *)(y[8*i]),
    _mm512_add_pd(_mm512_load_pd((void *)(y[8*i])),
    _mm512_mul_pd(_mm512_set1_pd(alpha),
    _mm512_load_pd((void *)(x[8*i])))
    ));
```
Vectorization on modern CPUs and MIC: AVX

- All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.
- 2012’ and newer CPU can process 4 doubles with one instruction (AVX).
- Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power.
- Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.
- ⇒ we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

```cpp
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```cpp
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void *)&y[8*i], _mm512_add_pd(_mm512_load_pd((void *)&y[8*i]),
               _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((void *)&x[8*i]))));
```
Vectorization on modern CPUs and MIC: AVX

- All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.
- 2012’ and newer CPU can process 4 doubles with one instruction (AVX).
- Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power.
- Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.
- ⇒ we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

Scalar loop:

```c
for (int i = 0; i < N; i++)
    y[i] += alpha * x[i];
```

Vectorized equivalent:

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void *) (y + 8*i),
                    _mm512_add_pd(_mm512_load_pd((void *) (y + 8*i)),
                                  _mm512_mul_pd(_mm512_set1_pd(alpha),
                                               _mm512_load_pd((void *) (x + 8*i))));
```
Vectorization on modern CPUs and MIC: AVX

- All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.
- 2012’ and newer CPU can process 4 doubles with one instruction (AVX).
- Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power.
- Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.
- ⇒ we need to develop vector-aware code, in order to make it efficient!

Example of scalar loop and its vectorized equivalent:

```c
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void*) (y[8*i]), _mm512_add_pd(_mm512_load_pd((void*) (y[8*i])),
    _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((void*) (x[8*i]))));
```
All modern CPUs have vector registers to process from 16 (SSE – old CPUs) to 64 (Xeon Phi) bytes of data simultaneously.  

2012’ and newer CPU can process 4 doubles with one instruction (AVX).  

Running code without vectorization ⇒ use of only ≈1/4 of CPU’s core compute power.  

Compilers try to vectorize scalar code automatically, but usually succeed in simple cases only.  

⇒ we need to develop vector-aware code, in order to make it efficient!

**Example of scalar loop and its vectorized equivalent:**

```c
for (int i = 0; i < N; i++) y[i] += alpha * x[i];
```

```c
for (int i = 0; i < N / 8; i++)
    _mm512_store_pd((void *)(y[8*i]), _mm512_add_pd(_mm512_load_pd((void *)(y[8*i])),
                   _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((void *)(x[8*i])))));
```
**Vectorization: MIC vs GPU**

**MIC:**
- 512-bit vectorization is absolutely necessary on MIC, similarly to how 32/128-byte coalescing is essential on GPUs
- Compiler is able to auto-vectorize simple loops, but in more complex cases it’s usually conservative
- It may be need to vectorize at least some parts of application *manually*
- On existing MIC cards vector instructions only work with aligned memory addresses

**GPU/GPU:**
- On GPU there is very little vector arithmetics, but there are vector memory load/store instructions
- Compiler tries to use vectorized loads/stores, when one thread accesses consecutive memory addresses *(Q: how it helps?)*
- Presence of LD.128/ST.128 instructions in disassembly *(cuobjdump -sass)* is an evidence of successful load/store vectorization
Vectorization

Scalar code:

```c
void ss_axpy(Field& y, const double alpha, Field const& x, const int N)
{
    for (int i = 0; i < N; i++) y[i] += alpha * x[i];
}
```

Vectorization for MIC:

```c
void ss_axpy(Field& y, const double alpha, Field const& x, const int N)
{
    #pragma omp parallel for
    for (int i = 0; i < N / 8; i++)
        _mm512_store_pd((__void*)(y.data() + 8*i), _mm512_add_pd(_mm512_load_pd((__void*)(y.data() + 8*i)), _mm512_mul_pd(_mm512_set1_pd(alpha), _mm512_load_pd((__void*)(x.data() + 8*i))));
}
```

Vectorization for GPU:

```c
template<short V, typename T>
__global__ void kernel(double* __restrict__ y, const double alpha, const double* __restrict__ x, const int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (V * i >= N) return;
    T yy, xx = T::ld(x, i), T zz = T::ld(y, i);
    for (int v = 0; v < V; v++)
        yy.v[v] = alpha * xx.v[v] + zz.v[v];
    T::stcs(y, i, yy);
}
```
Vectorization has effect on global memory throughput:

<table>
<thead>
<tr>
<th>version</th>
<th>memory throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>vectorized diffusion only</td>
<td>20.86 GB/sec</td>
</tr>
<tr>
<td>full vectorization</td>
<td>25.71 GB/sec</td>
</tr>
</tbody>
</table>

```bash
$ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced KMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
```
Vectorization: results on NVIDIA GPU for *wave13pt* test

- Vectorized code is 15% faster
- 2-element vectorization improves memory efficiency and reduces arithmetics (less indexing?)

![Figure: Naïve CUDA](image1.png)

![Figure: Vectorized CUDA](image2.png)

- Btw, this is an example of how code can turn from compute-bound into memory-bound as result of optimization! ☺
Vectorization: results on NVIDIA GPU for wave13pt test

- Vectorized code is 15% faster
- 2-element vectorization improves memory efficiency and reduces arithmetics (less indexing?)

![Figure: Naïve CUDA](image1.png)

![Figure: Vectorized CUDA](image2.png)

- Btw, this is an example of how code can turn from compute-bound into memory-bound as result of optimization! 😊
Global memory throughput is larger in vectorized version (very close to cuMemcpyDtoD’s, which gives 84 Gb/sec on this device).

Figure: Naïve CUDA

Figure: Vectorized CUDA

On linear algebra kernels of Application No.1 vectorization shows no improvement
Memory alignment

**GPU/GPU:**

- For scalar types unaligned accessing is not supported (“Warp Misaligned Address” error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaArray* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don’t do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
    for (int j = 2 + j_start; j < ny - 2; j += j_inc)
        for (int i = i_start; i < nx - 2; i += i_inc)
            if (i < 2) continue;
```

**CPU/MIC:**

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give “Segmentation fault”):

```c
_mm512_load_pd (void const* mem_addr);
_mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**
- For scalar types unaligned accessing is not supported (“Warp Misaligned Address” error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaArray* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don’t do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
  for (int j = 2 + j_start; j < ny - 2; j += j_inc)
    for (int i = i_start; i < nx - 2; i += i_inc)
      { if (i < 2) continue; }
```

**CPU/MIC:**
- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give “Segmentation fault”):
  - `_mm512_load_pd` and `_mm512_loadu_pd` commands

```c
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```
- `_mm512_loadu_pd` is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**
- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaMemcpy* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don't do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc) {
  for (int j = 2 + j_start; j < ny - 2; j += j_inc) {
    for (int i = i_start; i < nx - 2; i += i_inc) {
      if (i < 2) continue;
    }
  }
}
```

**CPU/MIC:**
- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give "Segmentation fault"):  
  - `__m512d _mm512_load_pd (void const* mem_addr);`
  - `__m512d _mm512_loadu_pd (void const* mem_addr);`
  - `_mm512_loadu_pd` is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**

- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaMemcpy* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don't do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
    for (int j = 2 + j_start; j < ny - 2; j += j_inc)
        for (int i = 1 + i_start; i < nx - 2; i += 1_inc)
            
            if (i < 2) continue;
```

**CPU/MIC:**

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give "Segmentation fault"): __m512d _mm512_load_pd (void const* mem_addr);
  __m512d _mm512_loadu_pd (void const* mem_addr);

- _mm512_loadu_pd is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**

- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with `cudaArray*` functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don't do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
    for (int j = 2 + j_start; j < ny - 2; j += j_inc)
        for (int i = 1_start; i < nx - 2; i += 1_inc)
            if (i < 2) continue;
```

**CPU/MIC:**

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give "Segmentation fault"): _mm512_loadu_pd is not supported by any existing Xeon Phi (KNL)

```c
_mm512_loadu_pd (void const* mem_addr);
_mm512_loadu_pd (void const* mem_addr);
```
Memory alignment

**GPU/GPU:**
- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with `cudaArray*` functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don't do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
  for (int j = 2 + j_start; j < ny - 2; j += j_inc)
    for (int i = 1 + i_start; i < nx - 2; i += i_inc)
      if (i < 2) continue;
```

**CPU/MIC:**
- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give "Segmentation fault"):
  ```c
  __m512d _mm512_load_pd (void const* mem_addr);
  __m512d _mm512_loadu_pd (void const* mem_addr);
  ```
- `_mm512_loadu_pd` is not supported by any existing Xeon Phi (KNL)
Memory alignment

GPU/GPU:

- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaMemcpy* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don’t do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
    for (int j = 2 + j_start; j < ny - 2; j += j_inc)
        for (int i = 1 + i_start; i < nx - 2; i += 1_inc)
            if (i < 2) continue;
```

CPU/MIC:

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give “Segmentation fault”):
  - \_mm512d _mm512_load_pd (void const* mem_addr);
  - \_mm512d _mm512_loadu_pd (void const* mem_addr);

- \_mm512\_loadu\_pd is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**

- For scalar types unaligned accessing is not supported ("Warp Misaligned Address" error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with `cudaArray*` functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don’t do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
    for (int j = 2 + j_start; j < ny - 2; j += j_inc)
        for (int i = 1 + i_start; i < nx - 2; i += i_inc)
            if (i < 2) continue;
```

**CPU/MIC:**

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give “Segmentation fault”):
  ```c
  __m512d _mm512_load_pd (void const* mem_addr);
  __m512d _mm512_loadu_pd (void const* mem_addr);
  ```
  _mm512_loadu_pd is not supported by any existing Xeon Phi (KNL)
Memory alignment

**GPU/GPU:**

- For scalar types unaligned accessing is not supported (“Warp Misaligned Address” error)
- For vector types vector-sized alignment is not required
- Performance is better for coalesced transaction aligned by the size of transaction:
  - Pad arrays manually or with cudaArray* functions
  - Use empty threads for boundaries, instead shrinking index range (only on GPU, don’t do this on CPU!):

```c
for (int k = 2 + k_start; k < ns - 2; k += k_inc)
  for (int j = 2 + j_start; j < ny - 2; j += j_inc)
    for (int i = i_start; i < nx - 2; i += i_inc)
      if (i < 2) continue;
```

**CPU/MIC:**

- Scalar values could be accessed with arbitrary aligning
- For vector types unaligned accessing is possible with a special (and more expensive) command (aligned command will give “Segmentation fault”):
  ```c
  __m512d_mm512_load_pd (void const* mem_addr);
  __m512d_mm512_loadu_pd (void const* mem_addr);
  ```
  
- `_mm512_loadu_pd` is not supported by any existing Xeon Phi (KNL)
The `__restrict__` keyword is a quick hint to perform input arrays caching in texture memory:

```
__global__ void kernel(const double* const __restrict__ up, double* __restrict__ sp)
```

<table>
<thead>
<tr>
<th>Without <code>__restrict__</code>:</th>
<th>With <code>__restrict__</code>:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transactions</strong></td>
<td><strong>Transactions</strong></td>
</tr>
<tr>
<td><strong>Bandwidth</strong></td>
<td><strong>Bandwidth</strong></td>
</tr>
<tr>
<td><strong>Utilization</strong></td>
<td><strong>Utilization</strong></td>
</tr>
<tr>
<td>L1/Shared Memory</td>
<td>L1/Shared Memory</td>
</tr>
<tr>
<td>Local Loads</td>
<td>1352959</td>
</tr>
<tr>
<td>Shared Loads</td>
<td>1436548</td>
</tr>
<tr>
<td>Shared Stores</td>
<td>23459432</td>
</tr>
<tr>
<td>Global Loads</td>
<td>2127810</td>
</tr>
<tr>
<td>Global Stores</td>
<td>18330931</td>
</tr>
<tr>
<td>L1/Shared Total</td>
<td>280031472</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>L2 Cache</td>
</tr>
<tr>
<td>Reads</td>
<td>605213592</td>
</tr>
<tr>
<td>Writes</td>
<td>1804739214</td>
</tr>
<tr>
<td>Total</td>
<td>785795516</td>
</tr>
<tr>
<td>Texture Cache</td>
<td>Texture Cache</td>
</tr>
<tr>
<td>Reads</td>
<td>5152888075</td>
</tr>
<tr>
<td>Writes</td>
<td>2882043311</td>
</tr>
<tr>
<td>Total</td>
<td>7241153308</td>
</tr>
<tr>
<td>Device Memory</td>
<td>Device Memory</td>
</tr>
<tr>
<td>Reads</td>
<td>5043610900</td>
</tr>
<tr>
<td>Writes</td>
<td>2058069607</td>
</tr>
<tr>
<td>Total</td>
<td>7102220867</td>
</tr>
<tr>
<td>System Memory</td>
<td>System Memory</td>
</tr>
<tr>
<td>Reads</td>
<td>168</td>
</tr>
<tr>
<td>Writes</td>
<td>202</td>
</tr>
<tr>
<td>Total</td>
<td>1700</td>
</tr>
<tr>
<td>System Memory</td>
<td></td>
</tr>
<tr>
<td>Reads</td>
<td>137</td>
</tr>
<tr>
<td>Writes</td>
<td>227</td>
</tr>
<tr>
<td>Total</td>
<td>364</td>
</tr>
</tbody>
</table>

It could be seen that texture memory partially takes the caching job. However, it has almost no effect on Application No.1 performance.
The __restrict__ keyword is a quick hint to perform input arrays caching in texture memory:

```
__global__ void kernel(const double* const __restrict__ up, double* __restrict__ sp)
```

Without __restrict__:

With __restrict__:

It could be seen that texture memory partially takes the caching job. However, it has almost no effect on Application No.1 performance.
Application No.1: Implementations

**GPU: NVIDIA Tesla K20c**
- Diffusion kernel in CUDA
- Linear algebra in CUDA
- Reducing synchronization overheads between CUDA kernels in CUDA kernels
- Vectorization of loads and stores
- Kernel specialization
- Dynamic parallelism: all GPU code as a single kernel

**GPU: Gigabyte Radeon HD 7970**
- Diffusion kernel in OpenCL
- Ddot & Daxpy – in clAmdBlas, all the rest – in OpenCL
- Reducing synchronization overheads between OpenCL kernels
- Kernel specialization

**MIC: Intel Xeon Phi 3120A**
- Diffusion and linear algebra in OpenMP
- Manual vectorization with AVX-512
- Loops unrolling
- Reducing synchronization overheads between OpenMP loops
- Use of huge memory pages
Reducing synchronization overheads: GPU

- **GPU**: The `cudaDeviceSynchronize` call is expensive, both on host and device (dynamic parallelism)
- **GPU**: The `clFlush` call is expensive

Optimization: Runtime can detect explicit data dependencies between kernels and perform synchronization automatically → in many cases user synchronization call could be removed, reducing overheads:

(Kernel times stay the same, e.g. `ss_axpy`: 179.653 µs → 178.021 µs.)
Reducing synchronization overheads: GPU

- **GPU:** The `cudaDeviceSynchronize` call is expensive, both on host and device (dynamic parallelism).
- **GPU:** The `clFlush` call is expensive.

Optimization: Runtime can detect explicit data dependencies between kernels and perform synchronization automatically ⇒ in many cases user synchronization call could be removed, reducing overheads:

(Kernel times stay the same, e.g. `ss_axpy`: 179.653 µs → 178.021 µs.)
Reducing synchronization overheads: MIC

- MIC: OpenMP runtime is in charge of threads synchronization, and it may take long, if workloads are imbalanced:
  - No affinity setting $\Rightarrow$ thread may freely migrate between cores. On CPU this has minor effect, because of L3 cache help. On MIC there is no L3 cache $\Rightarrow$ thread efficiency may degrade if it migrates too far from its cached data. The KMP_AFFINITY=granularity=fine,balanced pins logical threads to physical threads in cores:

- Xeon Phi runs Linux. It might be better to assign a separate core to OS with KMP_PLACE_THREADS=56c,4t:

  ```bash
  $ micnativeloadex ./main -a "512 512 400 0.01"
  simulation took 31.7525 seconds
  $ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced
  KMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
  simulation took 25.6407 seconds
  ```
Reducing synchronization overheads: MIC

- **MIC:** OpenMP runtime is in charge of threads synchronization, and it may take long, if workloads are imbalanced:
  - No affinity setting $\Rightarrow$ thread may freely migrate between cores. On CPU this has minor effect, because of L3 cache help. On MIC there is no L3 cache $\Rightarrow$ thread efficiency may degrade if it migrates too far from its cached data. The KMP_AFFINITY=granularity=fine,balanced pins logical threads to physical threads in cores:

  Xeon Phi runs Linux. It might be better to assign a separate core to OS with KMP_PLACE_THREADS=56c,4t:

  ```
  $ micnativeloadex ./main -a "512 512 400 0.01"
  simulation took 31.7525 seconds
  $ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced
  KMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
  simulation took 25.6407 seconds
  ```
Reducing synchronization overheads: MIC

- **MIC**: OpenMP runtime is in charge of threads synchronization, and it may take long, if workloads are imbalanced:
  - No *affinity* setting $\Rightarrow$ thread may freely migrate between cores. On CPU this has minor effect, because of L3 cache help. On MIC there is no L3 cache $\Rightarrow$ thread efficiency may degrade if it migrates too far from its cached data. The KMP_AFFINITY=granularity=fine,balanced pins logical threads to physical threads in cores:

- Xeon Phi runs Linux. It might be better to assign a separate core to OS with KMP_PLACE_THREADS=56c,4t:

```bash
$ micnativeloadex ./main -a "512 512 400 0.01"
simulation took 31.7525 seconds
```

```bash
$ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced ->
KMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
simulation took 25.6407 seconds
```
Reducing synchronization overheads: MIC

- **MIC**: OpenMP runtime is in charge of threads synchronization, and it may take long, if workloads are imbalanced:
  - No *affinity* setting $\Rightarrow$ thread may freely migrate between cores. On CPU this has minor effect, because of L3 cache help. On MIC there is no L3 cache $\Rightarrow$ thread efficiency may degrade if it migrates too far from its cached data. The `KMP_AFFINITY=granularity=fine,balanced` pins logical threads to physical threads in cores:

```
$ micnativeloadex ./main -a "512 512 400 0.01"
```

- Xeon Phi runs Linux. It might be better to assign a separate core to OS with `KMP_PLACE_THREADS=56c,4t`:

```
$ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced \nKMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
```

```bash
simulation took 31.7525 seconds
```

$ micnativeloadex ./main -a "512 512 400 0.01"
$ micnativeloadex ./main -e "KMP_AFFINITY=granularity=fine,balanced \nKMP_PLACE_THREADS=56c,4t" -a "512 512 400 0.01"
```

```
simulation took 25.6407 seconds
```
Application No.1: Implementations

**GPU: NVIDIA Tesla K20c**
- Diffusion kernel in CUDA
- Linear algebra in CUDA
- Reducing synchronization overheads between CUDA kernels in CUDA kernels
- Vectorization of loads and stores
- **Kernel specialization**
- Dynamic parallelism: all GPU code as a single kernel

**GPU: Gigabyte Radeon HD 7970**
- Diffusion kernel in OpenCL
- Ddot & Daxpy – in clAmdBlas, all the rest – in OpenCL
- Reducing synchronization overheads between OpenCL kernels
- **Kernel specialization**

**MIC: Intel Xeon Phi 3120A**
- Diffusion and linear algebra in OpenMP
- Manual vectorization with AVX-512
- Loops unrolling
- Reducing synchronization overheads between OpenMP loops
- Use of huge memory pages
void kernel(uchar4 pix1_r, uchar4 pix1_g, uchar4 pix1_b, uchar4 pix2_r, uchar4 pix2_g, uchar4 pix2_b,
    uchar4 pix3_r, uchar4 pix3_g, uchar4 pix3_b, int width, int height)
{
    int x = threadIdx.x + blockDim.x * blockIdx.x, y = threadIdx.y + blockDim.y * blockIdx.y;

    if ((x == width) || (y == 1) || (y == height - 1)) return;

    uchar4 r3 = pix3_r[x + y * width]; uchar4 g3 = pix3_g[x + y * width]; uchar4 b3 = pix3_b[x + y * width];

    if (y % 2 == 0) {
        uchar4 r1 = pix1_r[x + (y / 2) * width], g1 = pix1_g[x + (y / 2) * width], b1 = pix1_b[x + (y / 2) * width];

        r3.x = r1.x; r3.y = r1.y; r3.z = r1.z; r3.w = r1.w; g3.x = g1.x; g3.y = g1.y; g3.z = g1.z; g3.w = g1.w;
        b3.x = b1.x; b3.y = b1.y; b3.z = b1.z; b3.w = b1.w;
    } else {
        uchar4 r1 = pix1_r[x + (y / 2 - 1) * width], g1 = pix1_g[x + (y / 2 - 1) * width], b1 = pix1_b[x + (y / 2 - 1) * width];

        uchar4 r1b = pix1_r[x + (y / 2 + 1) * width], g1b = pix1_g[x + (y / 2 + 1) * width], b1b = pix1_b[x + (y / 2 + 1) * width];

        r1.x = (r1.x + r1b.x) / 2; r1.y = (r1.y + r1b.y) / 2; r1.z = (r1.z + r1b.z) / 2; r1.w = (r1.w + r1b.w) / 2;
        g1.x = (g1.x + g1b.x) / 2; g1.y = (g1.y + g1b.y) / 2; g1.z = (g1.z + g1b.z) / 2; g1.w = (g1.w + g1b.w) / 2;
        b1.x = (b1.x + b1b.x) / 2; b1.y = (b1.y + b1b.y) / 2; b1.z = (b1.z + b1b.z) / 2; b1.w = (b1.w + b1b.w) / 2;

        uchar4 r2 = pix2_r[x + (y / 2) * width], g2 = pix2_g[x + (y / 2) * width], b2 = pix2_b[x + (y / 2) * width];

        r3.x = (r1.x + r2.x) / 2; r3.y = (r1.y + r2.y) / 2; r3.z = (r1.z + r2.z) / 2; r3.w = (r1.w + r2.w) / 2;
        g3.x = (g1.x + g2.x) / 2; g3.y = (g1.y + g2.y) / 2; g3.z = (g1.z + g2.z) / 2; g3.w = (g1.w + g2.w) / 2;
        b3.x = (b1.x + b2.x) / 2; b3.y = (b1.y + b2.y) / 2; b3.z = (b1.z + b2.z) / 2; b3.w = (b1.w + b2.w) / 2;
    }
}
Kernel specialization: GPU

Split the kernel with condition on index into two unconditional kernels and run them one after another:

```c
__global__ void kernel1(
    uchar4* pix1_r, uchar4* pix1_g, uchar4* pix1_b,
    uchar4* pix2_r, uchar4* pix2_g, uchar4* pix2_b,
    uchar4* pix3_r, uchar4* pix3_g, uchar4* pix3_b, int width, int height)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = (threadIdx.y + blockIdx.y * blockDim.y) * 2;

    if (x >= width) return;

    pix3_r[x + y * width] = pix1_r[x + (y / 2) * width];
    pix3_g[x + y * width] = pix1_g[x + (y / 2) * width];
    pix3_b[x + y * width] = pix1_b[x + (y / 2) * width];
}
```
Split the kernel with condition on index into two unconditional kernels and run them one after another:

```c
__global__ void kernel2(uchar4* pix1_r, uchar4* pix1_g, uchar4* pix1_b, uchar4* pix2_r, uchar4* pix2_g, uchar4* pix2_b,
    uchar4* pix3_r, uchar4* pix3_g, uchar4* pix3_b, int width, int height)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x, y = (threadIdx.y + blockIdx.y * blockDim.y) * 2 + 3;

    if (x >= width) return;

    uchar4 r1 = pix1_r[x + (y / 2 - 1) * width], g1 = pix1_g[x + (y / 2 - 1) * width], b1 = pix1_b[x + (y / 2 - 1) * width];
    uchar4 r1b = pix1_r[x + (y / 2 + 1) * width], g1b = pix1_g[x + (y / 2 + 1) * width], b1b = pix1_b[x + (y / 2 + 1) * width];

    r1.x = (r1.x + r1b.x) / 2; r1.y = (r1.y + r1b.y) / 2; r1.z = (r1.z + r1b.z) / 2; r1.w = (r1.w + r1b.w) / 2;
    g1.x = (g1.x + g1b.x) / 2; g1.y = (g1.y + g1b.y) / 2; g1.z = (g1.z + g1b.z) / 2; g1.w = (g1.w + g1b.w) / 2;
    b1.x = (b1.x + b1b.x) / 2; b1.y = (b1.y + b1b.y) / 2; b1.z = (b1.z + b1b.z) / 2; b1.w = (b1.w + b1b.w) / 2;

    uchar4 r2 = pix2_r[x + (y / 2) * width], g2 = pix2_g[x + (y / 2) * width], b2 = pix2_b[x + (y / 2) * width];

    uchar4 r3 = pix3_r[x + y * width], g3 = pix3_g[x + y * width], b3 = pix3_b[x + y * width];

    r3.x = (r3.x + r2.x) / 2; r3.y = (r3.y + r2.y) / 2; r3.z = (r3.z + r2.z) / 2; r3.w = (r3.w + r2.w) / 2;
    g3.x = (g3.x + g2.x) / 2; g3.y = (g3.y + g2.y) / 2; g3.z = (g3.z + g2.z) / 2; g3.w = (g3.w + g2.w) / 2;
    b3.x = (b3.x + b2.x) / 2; b3.y = (b3.y + b2.y) / 2; b3.z = (b3.z + b2.z) / 2; b3.w = (b3.w + b2.w) / 2;
}
```

Result: execution time reduced from 0.826112 sec to 0.728256 sec.
Application No.1: Implementations

GPU: NVIDIA Tesla K20c
- Diffusion kernel in CUDA
- Linear algebra in CUDA
- Reducing synchronization overheads between CUDA kernels
- Vectorization of loads and stores
- Kernel specialization
- Dynamic parallelism: all GPU code as a single kernel

GPU: Gigabyte Radeon HD 7970
- Diffusion kernel in OpenCL
- Ddot & Daxpy – in clAmdBlas, all the rest – in OpenCL
- Reducing synchronization overheads between OpenCL kernels
- Kernel specialization

MIC: Intel Xeon Phi 3120A
- Diffusion and linear algebra in OpenMP
- Manual vectorization with AVX-512
- Loops unrolling
- Reducing synchronization overheads between OpenMP loops
- Use of huge memory pages
The use of huge pages is a solution to slow memory allocation problem on MIC: default malloc on MIC is much slower than GPU’s cudaMalloc
By default Linux uses 4 KB pages; huge page size is 2 MB

Method:
- Redefine malloc/free calls using mmap with MAP_HUGETLB flag

Result:
- Program doing simple array copying over memcpy became $2 \times$ faster

Link to Intel’s article with more details
**hddm-solver** – Adaptive Sparse Grids Solver for High-Dimensional Dynamic Models

(Johannes Brumm and Simon Scheidegger @ University of Zurich)

- Solves optimization problem in economics
- Evaluates the target function in the specified grid points
- Target function evaluation takes 90% of execution time

$$f(x) = \sin(\pi x) \cdot x^2 + 0.2(1-x)$$ with hierarchical linear basis functions of levels 1, 2, and 3.
1. Simplify arithmetic expressions, eliminate divisions (most expensive)
2. Eliminate duplicate computations, keeping the same byte per FLOP ratio, eliminate branching
3. Parallelize function evaluation with Thrust using combined transform+reduce (transform_reduce)
4. Eliminate redundant cudaMemcpy/cudaFree from Thrust implementation
5. Runtime optimization: hard-code vector size into GPU kernel and pass vector elements as scalars, together with other kernel arguments
   - 15% perf improvement, but needs JIT-compilation
   - Stores compiled kernels onto disk ⇒ could be slower on cluster FS, requires singleton for MPI/threads
6. Hybrid multithreading with Intel TBB: \((N - 1)\) threads on CPU, 1 thread - for GPU; TBB balances workloads automatically with “work stealing”
7. Vectorize CPU kernel with AVX
Eliminate redundant `cudaMalloc/cudaFree` from Thrust

- Thrust reduction requires scratch memory space (Q: why?)
  - For this reason, for each individual Thrust transform memory buffer is allocated & deallocated:

- Thrust does not allow to specify the scratch space. However, we can overload `cudaMalloc/cudaFree` in such way that memory would be allocated only once:

Result: Total execution time reduced from 216.27 sec to 201.80 s
Eliminate redundant `cudaMalloc/cudaFree` from Thrust

- Thrust reduction requires scratch memory space \(Q\): why?
- For this reason, for each individual Thrust transform memory buffer is allocated & deallocated:

Result: Total execution time reduced from 216.27 sec to 201.80 s
Eliminate redundant `cudaMalloc/cudaFree` from Thrust

- Thrust reduction requires scratch memory space (Q: why?)
- For this reason, for each individual Thrust transform memory buffer is allocated & deallocated:

```
Thrust does not allow to specify the scratch space. However, we can overload
`cudaMalloc/cudaFree` in such way that memory would be allocated only once:
```

Result: Total execution time reduced from 216.27 sec to 201.80 s
Eliminate redundant `cudaMalloc/cudaFree` from Thrust

- Thrust reduction requires scratch memory space (Q: why?)
- For this reason, for each individual Thrust transform memory buffer is allocated & deallocated:

Thrust does not allow to specify the scratch space. However, we can overload `cudaMalloc/cudaFree` in such way that memory would be allocated only once:

Result: Total execution time reduced from 216.27 sec to 201.80 s
Runtime optimization

- **GPU**: add input read-only vector elements as scalars to kernel arguments, which saves on overhead of a separate cudaMemcpy call:

```c
kernel<<<grid,block>>>(..., x[0], x[1], ..., x[DIM - 1]);
```

- **GPU** and **CPU**: in kernel code assemble scalars back into a local array (to ease indexing), which will likely be optimized out by compiler (full loop unrolling):

```c
const double x[] = { X_VALUES };
// This loop will likely be fully unrolled for a known small DIM.
for (int j = 0, vj = 0; j < DIM; j += 4, vj++)
{
    double4 x_4 = *(double4*)&x[j];
    ...
}
```

Requirements: Needs macros to manipulate the vector, code to invoke compiler from the running program and to load the compiled object (dlopen/dlsym).
Threading Building Blocks: workload balancing

- TBB maps independent tasks onto threads, similarly to OpenMP
  - Every thread is initially assigned an equal logical queue of tasks
  - However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores
  - TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress
  - hddm-solver maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths

Dmitry Mikushin et al. (Applied Parallel Computing LLC)
Threading Building Blocks: workload balancing

- TBB maps independent tasks onto threads, similarly to OpenMP
- Every thread is initially assigned an equal logical queue of tasks
  - However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores
- TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress
- hddm-solver maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths

![Diagram showing workload balancing in TBB](image-url)
**Threading Building Blocks: workload balancing**

- TBB maps independent tasks onto threads, similarly to OpenMP.
- Every thread is initially assigned an equal logical queue of tasks.
- However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores.

  - TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress.
  - `hddm-solver` maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths.
Threading Building Blocks: workload balancing

- TBB maps independent tasks onto threads, similarly to OpenMP.
- Every thread is initially assigned an equal logical queue of tasks.
- However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores.
- TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress.
- hddm-solver maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths.
Threading Building Blocks: workload balancing

- TBB maps independent tasks onto threads, similarly to OpenMP.
- Every thread is initially assigned an equal logical queue of tasks.
- However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores.
- TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress.
- hddm-solver maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths.
Threading Building Blocks: workload balancing

- TBB maps independent tasks onto threads, similarly to OpenMP
- Every thread is initially assigned an equal logical queue of tasks
- However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores
- TBB approach to work balancing: once one thread runs out of tasks, “steal” a task from another thread, which makes slower progress
- hddm-solver maps one extra thread onto GPU ⇒ CPU cores and GPU handle function evaluation tasks together, using different code paths
Application No.2: Optimization results

(less is better)
1. Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

2. Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good.

3. Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder).

4. MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance.

5. Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well.
1. Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

2. Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good.

3. Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder).

4. MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance.

5. Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well.
1. Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

2. Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good

3. Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder)

4. MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance

5. Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well
Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good.

Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder).

MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance.

Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well.
1 Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

2 Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good

3 Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder)

4 MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance

5 Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well
1. Better to purchase specific accelerator, already knowing the target application and its behavior on a test system (which we can give you access to!)

2. Primarily targeting accelerator, it still makes sense to develop hybrid solution as in Application No.2, if CPU is also good

3. Coalescing on GPU/GPU at worst requires coding effort comparable to vectorization on MIC (overall, vectorization seems to be harder)

4. MIC real memory throughput is not so good as in GPU and GPU, resulting into worse memory-bound application performance

5. Profilers for GPU and GPU generally give very useful info, but sometimes do not behave well