

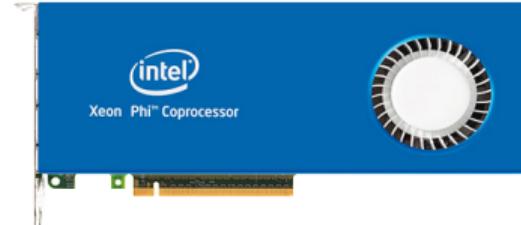


## Side by side maximization of GPU/MIC accelerators performance by example of numerical simulation problem

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

July 2, 2014

# Три типа современных ускорителей



GPU: NVIDIA Tesla K20c

Kepler GK110, 28 nm

13 мп × 192 ядра @ 0.71 GHz

5 GB GDDR5 @ 2.6 GHz

225W

ECC: есть

74736 RUR

GPU: Gigabyte Radeon HD 7970

Graphics Core Next, 28 nm

32 мп × 64 ядра @ 1 GHz

3GB GDDR5 @ 1.5 GHz

250W

ECC: нет

13000 RUR

MIC: Intel Xeon Phi 3120A

Knights Corner (KNC), 22 nm

57 ядер @ 1.1 GHz

6GB GDDR5 @ 1.1 GHz

300W

ECC: есть

60602 RUR

до 4 потоков на ядро

512-битные векторные  
инструкции (AVX-512)

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

- Измерение времени в сравнении с CPU-версией
  - Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺
- Сравнение с производительностью стандартных библиотек
  - Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)
- Детальный анализ целевого приложения
  - Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
  - Анализ достижимых характеристик ускорителя
  - Профилирование

- Измерение времени в сравнении с CPU-версией
  - Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺
- Сравнение с производительностью стандартных библиотек
  - Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)
- Детальный анализ целевого приложения
  - Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
  - Анализ достижимых характеристик ускорителя
  - Профилирование

- Измерение времени в сравнении с CPU-версией
  - Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺
- Сравнение с производительностью стандартных библиотек
  - Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)
- Детальный анализ целевого приложения
  - Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
  - Анализ достижимых характеристик ускорителя
  - Профилирование

# Три способа анализа эффективности приложений на ускорителях

- Измерение времени в сравнении с CPU-версией
  - Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺
- Сравнение с производительностью стандартных библиотек
  - Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)
- Детальный анализ целевого приложения
  - Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
  - Анализ достижимых характеристик ускорителя
  - Профилирование

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

# Использование стандартных библиотек для анализа эффективности

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

Использование при анализе:

- 1 Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- 2 В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- 3 Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

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

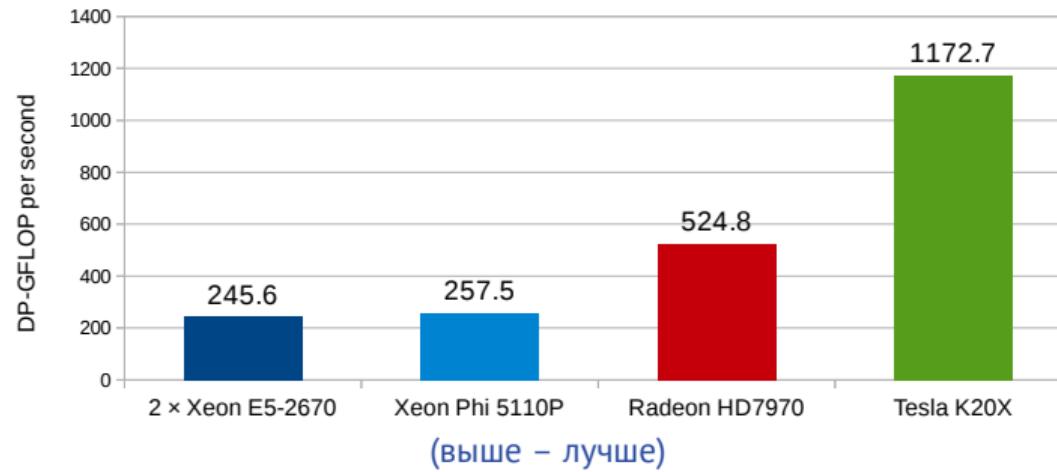
Использование при анализе:

- 1 Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- 2 В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- 3 Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)

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

Использование при анализе:

- 1 Можно быстро и точно оценить производительность приложения, использующего стандартную библиотеку
- 2 В сравнении со стандартной библиотекой можно оценить эффективность рукописных ядер (где они необходимы)
- 3 Можно оценить, какой ускоритель наиболее перспективно использовать для данной задачи (примеры – в следующих слайдах)



железо	#ядер	метод	компилятор
Xeon E5-2670 × 2	8 × 2	MKL	icpc -g -O3 -xHost
Xeon 5110P	60	MKL, native	icpc -g -O3 -mmic
Radeon HD7970	2,048	clAmdBlas	g++ -g -O3
Kepler K20X	2,496	CUBLAS	g++ -g -O3

icpc/mkl: 2013.2.146, cublas: 5.5

## CPU BLAS

```
#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

```
#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;
cUBLAS_Dgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
              n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

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

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cUBLAS_Destroy(handle);
```

# BLAS DGEMM

## CPU BLAS

```
#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

```
#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

```
#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

```
#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

```
#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

```
#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

```
#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

```
#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;
cUBLAS_Dgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
              n, n, n, &alpha, A_dev, n, B_dev, n, &beta, C_dev, n);
cudaDeviceSynchronize();

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

cudaFree(A_dev);
cudaFree(B_dev);
cudaFree(C_dev);
cUBLAS_Destroy(handle);
```

# BLAS DGEMM

## CPU BLAS

```
#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

```
#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

```
#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

```
#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

```
#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);
```

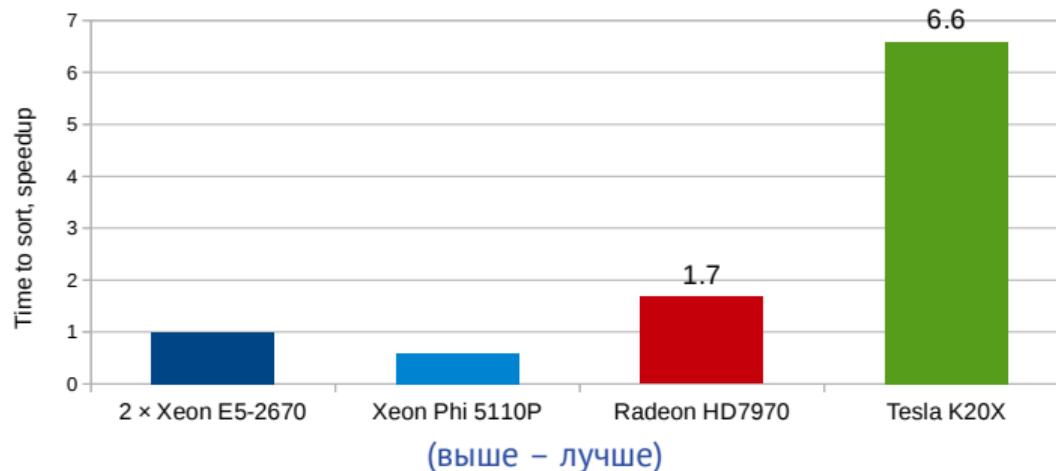
## GPU-enabled BLAS

В CUDA 6.0 добавлена библиотека **NVBLAS**:

- Автоматически перенаправляет вызовы BLAS 3-го уровня в CUBLAS (LD\_PRELOAD=libnvblas.so)
- Может распределять вычисления между несколькими GPU и CPU (cuBLAS-XT)
- Может применяться в любых приложениях, использующих BLAS3 (Octave, Scilab, ...)

```
double alpha = 1.0, beta = 0.0;  
dgemm_( 'n', 'n',  
         &n, &n, &n, &alpha, A, &n, B, &n, &beta, C, &n);
```

# Сортировка 128 млн пар ключ-значение



icpc: 2013.2.146, tbb: 4.1, bolt: 1.1, thrust: 5.5

# Сортировка 128 млн пар ключ-значение

```
#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());
```

# Сортировка 128 млн пар ключ-значение

```
#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();
}

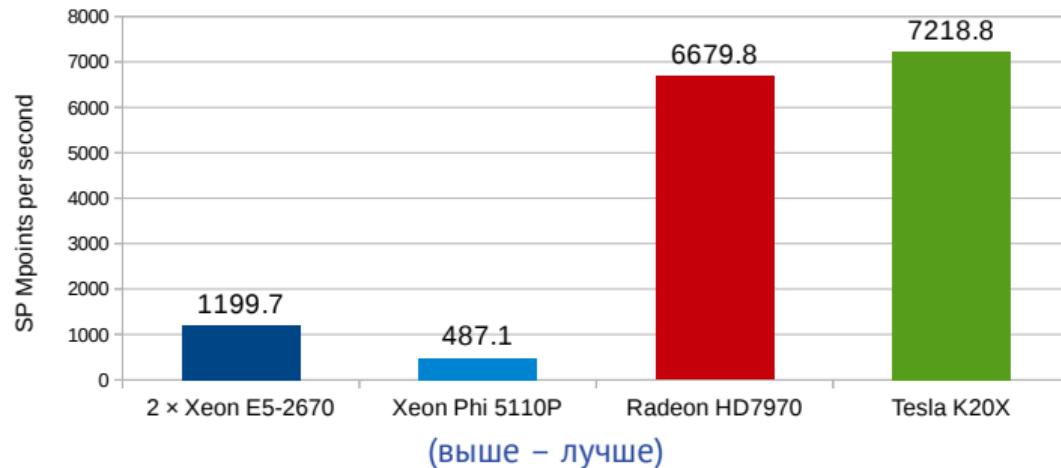
// Transfer data to the device.
device_vector<float> d_keys = h_keys;
device_vector<float> d_vals = h_vals;

// 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());
```

С помощью библиотеки CUB можно сделать radix-сортировку ещё в 2.5 раза быстрее!

# Численная схема для волнового уравнения



icpc: 2013.2.146, pgi: 14.1 – nvidia, 14.6 – radeon

# Численная схема для волнового уравнения

```
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));
            }
        }
    }
}
```

# Численная схема для волнового уравнения

```
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));
            }
        }
    }
}
```

OpenACC позволяет выгрузить вычисления на GPU с высокой эффективностью:

- PGI OpenACC 14.1 часто генерирует более быстрые ядра, чем написанные вручную на CUDA

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

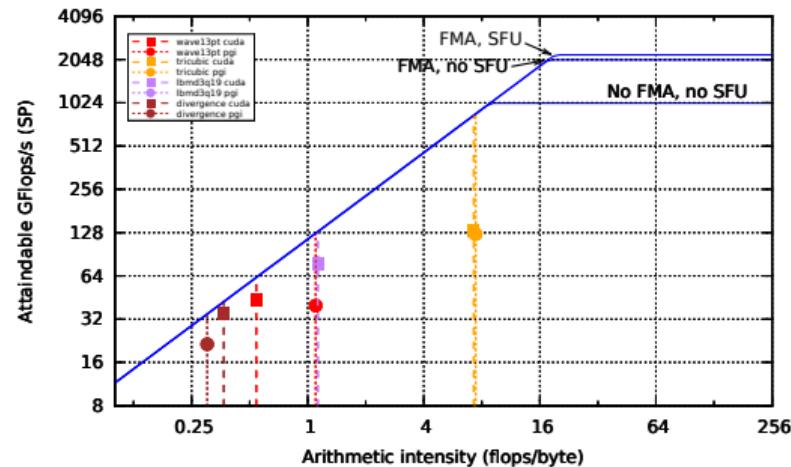
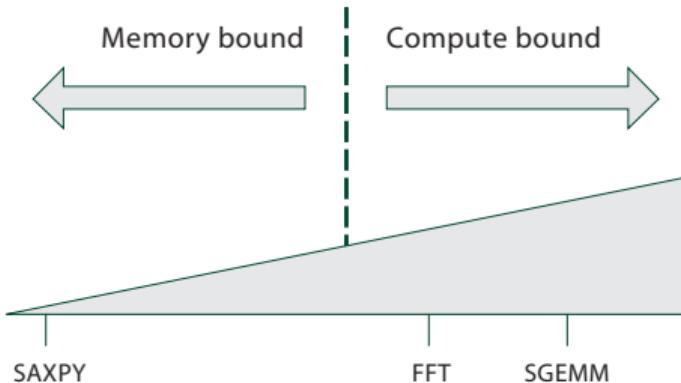
- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

# Compute-bound VS memory-bound

- **Compute-bound** – приложение даёт высокую нагрузку на вычислительные блоки процессора, но память использует менее эффективно
- **Memory-bound** – приложение интенсивно работает с глобальной памятью (DRAM, не кеш), вычислительные блоки простаивают, т.к. не обеспечены данными



- Если класс программы не предопределен теорией, то compute-bound может переходить в memory-bound и обратно в результате оптимизаций :)

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- **Анализ достижимых характеристик ускорителя**
- Профилирование

# Анализ пропускной способности глобальной памяти

- Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:
  - Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)
  - Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)
  - Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

■ Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)
- Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)
- Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

■ Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)
- Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)
- Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

- Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)
  - Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

- Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)

- Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

- Производитель может писать какой-угодно *memory bandwidth* в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если *memory-bound* приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если *memory-bound* приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)

Кеширование (в регистрах, в разделяемой памяти **GPU/GPU**)

- Если *memory-bound* приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

# Анализ пропускной способности глобальной памяти

- Производитель может писать какой-угодно memory bandwidth в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если memory-bound приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если memory-bound приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)

Кеширование (в регистрах, в разделяемой памяти GPU/GPU)

Вычисления «на лету»

- Если memory-bound приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

- Производитель может писать какой-угодно memory bandwidth в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если memory-bound приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если memory-bound приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)

Кеширование (в регистрах, в разделяемой памяти GPU/GPU)

Вычисления «на лету»

- Если memory-bound приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

- Производитель может писать какой-угодно memory bandwidth в характеристиках – практически полезно лишь то значение, которое достигается программно на простых тестах:

- Если memory-bound приложение создаёт высокую нагрузку на память, то это ещё не значит, что пропускная способность используется эффективно (что может этому мешать, например?)

Плохой коалесинг (потоку нужно 8 байт – он читает 128 и «выбрасывает» 120, т.к. они не нужны соседним потокам)

- Если memory-bound приложение использует пропускную способность памяти эффективно, то наибольший смысл имеют оптимизации, направленные на уменьшение числа обращений к памяти (какие, например?)

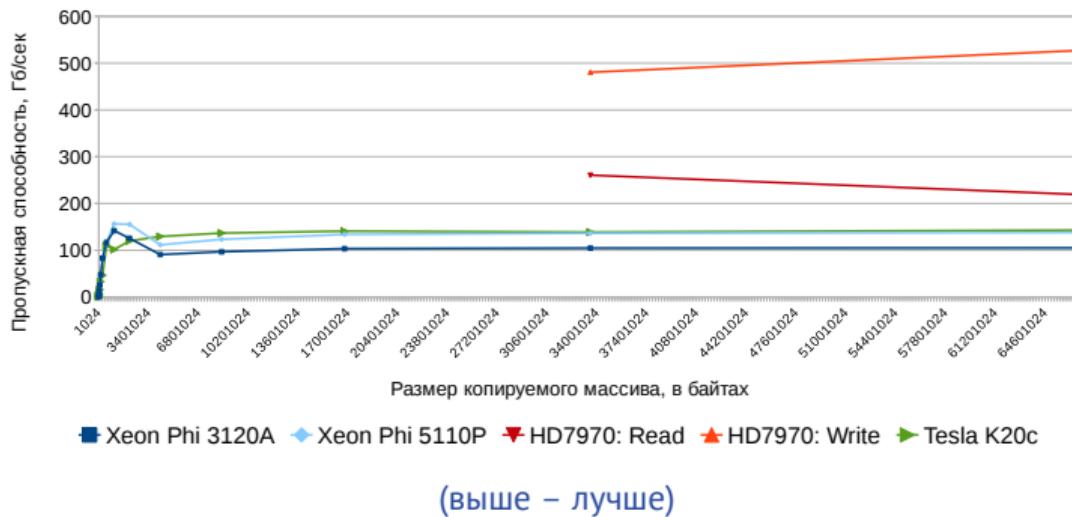
Кеширование (в регистрах, в разделяемой памяти GPU/GPU)

Вычисления «на лету»

- Если memory-bound приложение не использует всей пропускной способности, то возможно оно неоптимально (примеры причин?)

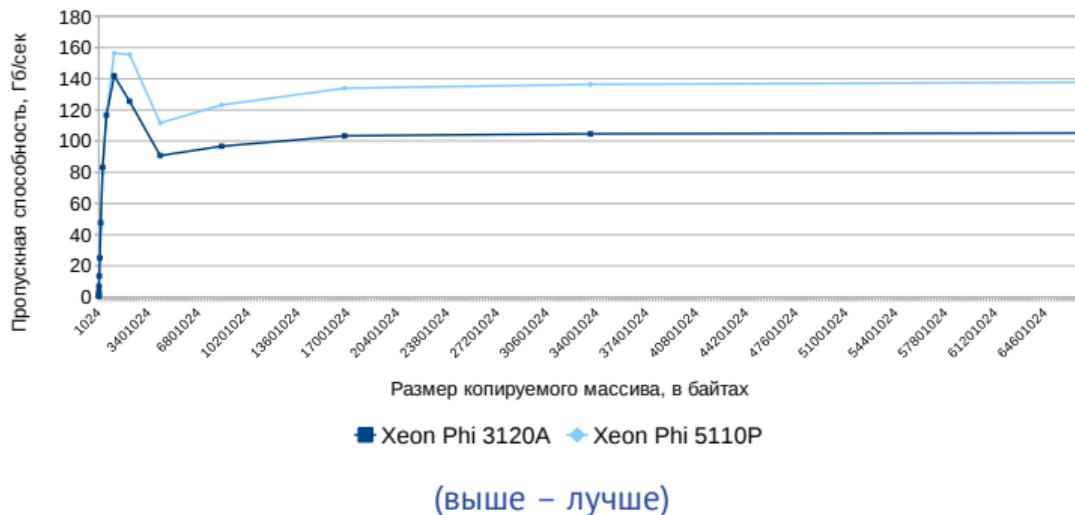
Недостаточно параллелизма (мало блоков (GPU/GPU/) или потоков (GPU/GPU/MIC))

# Анализ пропускной способности глобальной памяти



железо	#ядер	метод
Xeon 5110P	60	STREAM benchmark
Xeon 3120A	57	STREAM benchmark
Radeon HD7970	2,048	APPSDK
Kepler K20X	2,496	CUDA SDK / bandwidthTest

# Анализ пропускной способности памяти: различия между поколениями MIC



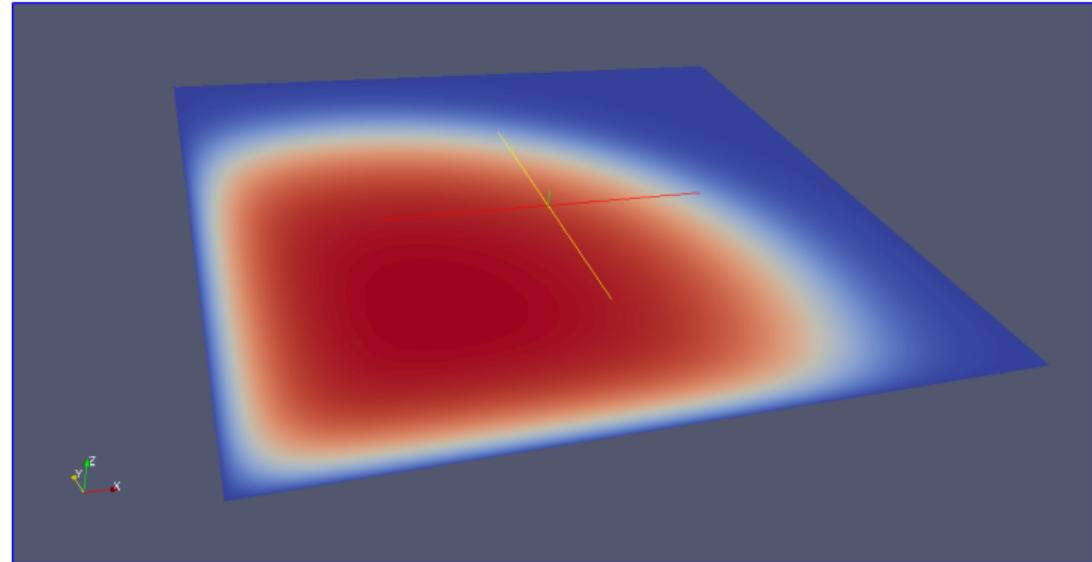
- Всего на MIC 8 контроллеров памяти, каждый с 2 32-битными каналами
- На карте 3120 4 канала заблокировано, работает только 12
- 3120: 32 бита  $\times$  12 каналов  $\times$  2,5ГГц  $\times$  2 стороны передачи = **240 Гб/сек**
- 5110: 32 бита  $\times$  16 каналов  $\times$  2,5ГГц  $\times$  2 стороны передачи = **320 Гб/сек**

# Приложение №1

Нелинейное уравнение  
диффузии-реакции  
(уравнение Фишера):

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

- Прямоугольная расчётная область
- Границные условия Дирихле



# Приложение №1: Алгоритм

```
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:   tolerance  $\leftarrow 10^{-6}$ 
7:   итерации по времени:
8:   for  $i = 1, i_{max}$  do
9:      $x_{old} \leftarrow x_{new}$ 
10:  нелинейные итерации:
11:    for  $j = 1, j_{max}$  do
12:      diffusion( $x_{new}, x_{old}, b$ );
13:      if  $\|b\| < tolerance$  then break
14:       $is\_converged \leftarrow cg(N, \delta x, b, cg_{max}, tolerance)$ 
15:      if not  $is\_converged$  then error
16:       $x_{new} \leftarrow x_{new} - \delta x;$ 
```

- Внешние итерации  $i$  по временным шагам
- Внутренние итерации  $j$  по нелинейности
- Применение оператора диффузии
- Решение СЛАУ методом сопряжённых градиентов

# Приложение №1: Алгоритм

```
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:   tolerance  $\leftarrow 10^{-6}$ 
7:   итерации по времени:
8:   for  $i = 1, i_{max}$  do
9:      $x_{old} \leftarrow x_{new}$ 
10:  нелинейные итерации:
11:    for  $j = 1, j_{max}$  do
12:      diffusion( $x_{new}, x_{old}, b$ );
13:      if  $\|b\| < tolerance$  then break
14:       $is\_converged \leftarrow cg(N, \delta x, b, cg_{max}, tolerance)$ 
15:      if not  $is\_converged$  then error
16:       $x_{new} \leftarrow x_{new} - \delta x;$ 
```

- Внешние итерации  $i$  по временным шагам
- Внутренние итерации  $j$  по нелинейности
- Применение оператора диффузии
- Решение СЛАУ методом сопряжённых градиентов

# Приложение №1: Алгоритм

```
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:   tolerance  $\leftarrow 10^{-6}$ 
7:   итерации по времени:
8:   for  $i = 1, i_{max}$  do
9:      $x_{old} \leftarrow x_{new}$ 
10:  нелинейные итерации:
11:    for  $j = 1, j_{max}$  do
12:      diffusion( $x_{new}, x_{old}, b$ );
13:      if  $\|b\| < tolerance$  then break
14:       $is\_converged \leftarrow cg(N, \delta x, b, cg_{max}, tolerance)$ 
15:      if not  $is\_converged$  then error
16:       $x_{new} \leftarrow x_{new} - \delta x;$ 
```

- Внешние итерации  $i$  по временным шагам
- Внутренние итерации  $j$  по нелинейности
- Применение оператора диффузии
- Решение СЛАУ методом сопряжённых градиентов

# Приложение №1: Алгоритм

```
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:   tolerance  $\leftarrow 10^{-6}$ 
7:   итерации по времени:
8:   for  $i = 1, i_{max}$  do
9:      $x_{old} \leftarrow x_{new}$ 
10:  нелинейные итерации:
11:    for  $j = 1, j_{max}$  do
12:      diffusion( $x_{new}, x_{old}, b$ );
13:      if  $\|b\| < tolerance$  then break
14:       $is\_converged \leftarrow cg(N, \delta x, b, cg_{max}, tolerance)$ 
15:      if not  $is\_converged$  then error
16:       $x_{new} \leftarrow x_{new} - \delta x;$ 
```

- Внешние итерации  $i$  по временным шагам
- Внутренние итерации  $j$  по нелинейности
- Применение оператора диффузии
- Решение СЛАУ методом сопряжённых градиентов

# Приложение №1: Алгоритм

```
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:   tolerance  $\leftarrow 10^{-6}$ 
7:   итерации по времени:
8:   for  $i = 1, i_{max}$  do
9:      $x_{old} \leftarrow x_{new}$ 
10:  нелинейные итерации:
11:    for  $j = 1, j_{max}$  do
12:      diffusion( $x_{new}, x_{old}, b$ );
13:      if  $\|b\| < tolerance$  then break
14:       $is\_converged \leftarrow cg(N, \delta x, b, cg_{max}, tolerance)$ 
15:      if not  $is\_converged$  then error
16:       $x_{new} \leftarrow x_{new} - \delta x;$ 
```

- Внешние итерации  $i$  по временным шагам
- Внутренние итерации  $j$  по нелинейности
- Применение оператора диффузии
- Решение СЛАУ методом сопряжённых градиентов

# Приложение №1: Реализация



**GPU:** NVIDIA Tesla K20c

- Ядро диффузии на CUDA
- Ядра функций линейной алгебры на CUDA
- Уменьшение синхронизаций (ожидания завершения CUDA-ядер)
- Векторизация LD/ST
- Специализация ядер
- Динамический параллелизм: весь алгоритм как одно GPU-ядро



**GPU:** Gigabyte Radeon HD 7970

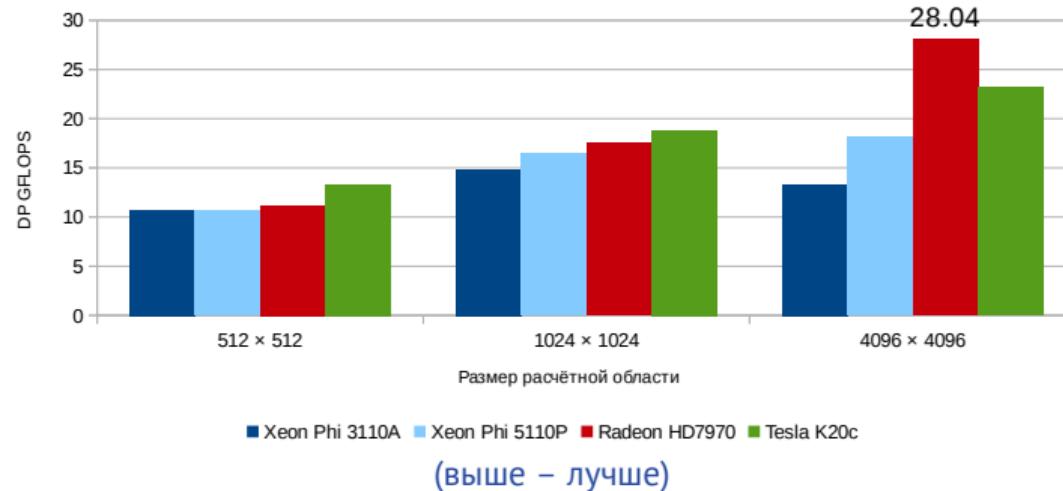
- Ядро диффузии на OpenCL
- Ddot и Daxpy – из clAmdBlas, остальное – OpenCL
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Специализация ядер



**MIC:** Intel Xeon Phi 3120A

- Диффузия и лин. алгебра – на OpenMP
- Векторизация вычислений вручную с помощью AVX-512
- Раскрутка циклов
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Использование больших страниц памяти

# Приложение №1: Результаты



размер	Xeon Phi 3110A	Xeon Phi 5110P	Radeon HD7970	Tesla K20c
512 <sup>2</sup>	10.714055	10.588785	11.189923	13.302244
1024 <sup>2</sup>	14.861325	16.489451	17.483508	18.846611
4096 <sup>2</sup>	13.306116	18.133706	28.035458	23.144442

# Три способа анализа эффективности приложений на ускорителях

## ■ Измерение времени в сравнении с CPU-версией

- Уровень оптимизации исходной версии не всегда принимается во внимание  
⇒ хороший способ получить ускорение в 100 раз и более ☺

## ■ Сравнение с производительностью стандартных библиотек

- Проверка производительности на достижение известных типовых показателей (например, 300 GFLOPS на DP DGEMM для Tesla C2075)

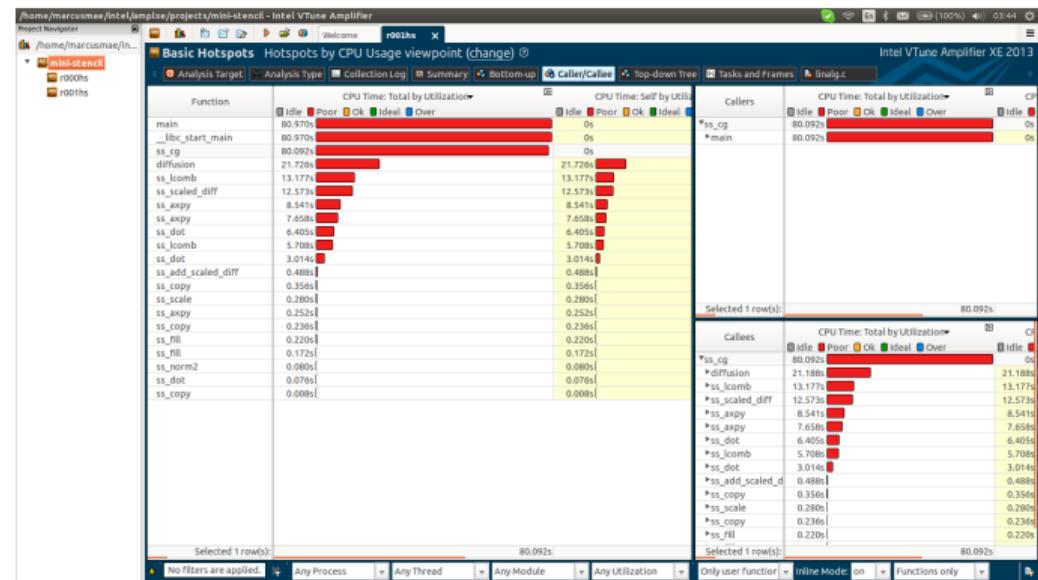
## ■ Детальный анализ целевого приложения

- Использование нормированных метрик: FLOPS, FLOP/byte (roofline)
- Анализ достижимых характеристик ускорителя
- Профилирование

# Профилирование: CPU/MIC

## Intel VTune Amplifier

- Позволяет быстро от профилировать CPU-приложение для планирования переноса на ускоритель
- Предопределённые сценарии анализа удобны в использовании
- Профилирует также Xeon Phi (требуется установить на Xeon Phi модуль ядра)
- Метрика CPI Rate из Advanced Hostspots похожа на FLOPS/byte
- Показывает многопоточную эффективность приложения
- Бесплатен для некоммерческого использования при установке под Linux



# Профилирование: счётчики MIC в VTune

/home/ashevchenko/mic/vtune project/summer-school-mic - Intel VTune Amplifier

Project Navigator

Hardware Event Counts viewpoint (change) ?

Collection Log Analysis Target Analysis Type Summary PMU Events Caller/Callee Top-down Tree Tasks and Frames

Intel VTune Amplifier XE 2013

Grouping: Source Function Stack

Source Function Stack	Hardware Event Count: Total by Hardware Event Type			Hardware Event Count: Self by Hardware Event Type		Function (Full)	Source File
	INSTRUCTIONS_EXECUTED	CPU_CLK_UNHALTED	INSTRUCTIONS_EXECU...	CPU_CLK_UNHALTED	Function (Full)		
▼ Total	10,040,000,000	54,410,000,000	0	0	0		
► [libiomp5.so]	7,030,000,000	39,050,000,000	7,030,000,000	39,050,000,000	[libiomp ...		
► linalg:ss_cg\$omp\$parallel_for@140	480,000,000	1,930,000,000	480,000,000	1,930,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@97	420,000,000	1,800,000,000	420,000,000	1,770,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@97	380,000,000	1,810,000,000	380,000,000	1,790,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@119	370,000,000	1,420,000,000	370,000,000	1,420,000,000	linalg:ss ... linalg.cpp		
► [vmlinux]	350,000,000	1,120,000,000	350,000,000	1,120,000,000	[vmlinux]		
► linalg:ss_cg\$omp\$parallel_for@140	340,000,000	1,390,000,000	340,000,000	1,390,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@58	260,000,000	2,090,000,000	240,000,000	1,310,000,000	linalg:ss ... linalg.cpp		
► operators:diffusion\$omp\$parallel@64	100,000,000	1,320,000,000	100,000,000	1,250,000,000	operators ... operators ...		
► main	90,000,000	1,250,000,000	90,000,000	1,250,000,000	main	main.cpp	
► linalg:ss_cg\$omp\$parallel_for@149	30,000,000	190,000,000	30,000,000	190,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_axpy\$omp\$parallel_for@97	30,000,000	90,000,000	30,000,000	90,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@58	60,000,000	260,000,000	20,000,000	210,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@129	20,000,000	60,000,000	20,000,000	60,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@108	20,000,000	220,000,000	20,000,000	220,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@149	20,000,000	140,000,000	20,000,000	140,000,000	linalg:ss ... linalg.cpp		
► linalg:ss_cg\$omp\$parallel_for@83	20,000,000	60,000,000	20,000,000	60,000,000	linalg:ss ... linalg.cpp		
► [Import thunk std::ios_base::Init::~Init]	10,000,000	0	10,000,000	0	[Import ...		
► linalg:ss_cg	10,000,000	20,000,000	0	0	linalg:ss ... linalg.cpp		
► func@0x4018f3	0	20,000,000	0	20,000,000	func@0 ...		
► linalg:ss_norm2\$omp\$parallel_for@71	0	70,000,000	0	70,000,000	linalg:ss ... linalo.cpp		
	Selected 1 row(s):	10,040,000,000	54,410,000,000	0	0		

Q Q+ Q- Q= Q=

5s 10s 15s 20s 25s 30s 35s 40s 45s 50s 55s

main (TID: 44578)

thread

Thread

Running

Hardware Event Count

Dmitry Mikushin et al. (APC LLC & University of Lugano)

GPU/APU/MIC optimization

July 2, 2014

23 / 46

# Профилирование: счётчики MIC в VTune

/home/ashevchenko/mic/vtune project/summer-school-mic - Intel VTune Amplifier

Project Navigator

Hardware Event Counts viewpoint (change)

Collection Log Analysis Target Analysis Type Summary PMU Events Caller/Callee Top-down Tree Tasks and Frames

Intel VTune Amplifier XE 2013

Grouping: Call Stack

Function Stack	Hardware Event Count: Total by Hardware Event Type		Hardware Event Count: Self by Hardware Event Type		Mo.	Function (Full)	Source File	Sta.. Add.
	INSTRUCTIONS_EXECUTED	CPU_CLK_UNHALTED	INSTRUCTIONS_EXECU...	CPU_CLK_UNHALTED				
▼ Total	6,060,000,000	30,830,000,000	0	0				
↳ [libiomp5.so]	3,130,000,000	19,000,000,000	3,130,000,000	19,000,000,000	libi...	[libiomp...	0	
↳ linalg:ss_cg\$omp\$parallel_for	570,000,000	1,730,000,000	570,000,000	1,730,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	530,000,000	1,820,000,000	530,000,000	1,820,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	500,000,000	1,770,000,000	500,000,000	1,770,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	490,000,000	1,500,000,000	490,000,000	1,500,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	470,000,000	1,530,000,000	470,000,000	1,530,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ operators::diffusion\$omp\$par	230,000,000	2,480,000,000	230,000,000	2,480,000,000	main	operator... operators...	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	40,000,000	160,000,000	40,000,000	160,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ [vmlinux]	20,000,000	300,000,000	20,000,000	300,000,000	vml...	[vmlinux]	0	
↳ linalg:ss_cg\$omp\$parallel_for	20,000,000	70,000,000	20,000,000	70,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_add_scaled_diff	20,000,000	70,000,000	20,000,000	70,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	20,000,000	50,000,000	20,000,000	50,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	10,000,000	110,000,000	10,000,000	110,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_norm2\$omp\$parallel_for	10,000,000	60,000,000	10,000,000	60,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ main	0	10,000,000	0	10,000,000	main	main.cpp	0x4...	
↳ linalg:ss_cg	0	20,000,000	0	10,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ __sti_SE	0	30,000,000	0	0	main	__sti_SE	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	0	20,000,000	0	20,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	0	30,000,000	0	30,000,000	main	linalg:ss... linalg.cpp	0x4...	
↳ linalg:ss_cg\$omp\$parallel_for	0	20,000,000	0	20,000,000	main	linalg:ss... linalg.cpp	0x4...	
Selected 1 row(s):	570,000,000	1,730,000,000	570,000,000	1,730,000,000				

Q Q+Q-Q Thread

1s 2s 3s 4s 5s 6s 7s 8s 9s 10s 11s 12s 13s 14s 15s 16s 17s 18s 19s 20s 21s 22s 23s 24s 25s 26s 27s 28s 29s

Running

Hardware Event Count

CPU\_CLK\_UNH/

Thread

Running

Hardware Event Count

CPU\_CLK\_UNH/

Dmitry Mikushin et al. (APC LLC & University of Lugano)

GPU/APU/MIC optimization

July 2, 2014

24 / 46

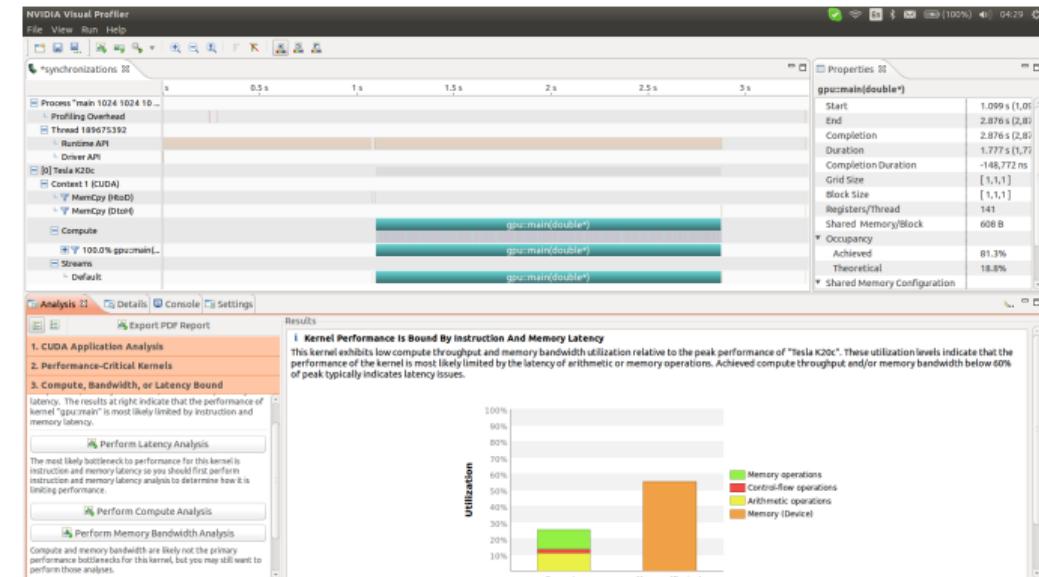
# Профилирование: NVIDIA

## NVIDIA Visual Profiler

- Графический профайлер на основе Eclipse
- Профилирует на локальном GPU, начиная с CUDA 6.0 умеет профилировать также на удалённом GPU (что очень и очень удобно!)
- Группирует результаты анализа
- В программе можно задать свои собственные регионы для профилирования

## Недостатки:

- Основан на Eclipse ⇒ начинает тормозить при первой удобной возможности
- В ходе профилирования приложение запускается большое число раз
- При анализе больших приложений время готовности результатов → ∞



# Приложение №1: Реализация



**GPU:** NVIDIA Tesla K20c

- Ядро диффузии на CUDA
- Ядра функций линейной алгебры на CUDA
- Уменьшение синхронизаций (ожидания завершения CUDA-ядер)
- **Векторизация LD/ST**
- Специализация ядер
- Динамический параллелизм: весь алгоритм как одно GPU-ядро



**GPU:** Gigabyte Radeon HD 7970

- Ядро диффузии на OpenCL
- Ddot и Daxpy – из clAmdBlas, остальное – OpenCL
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Специализация ядер



**MIC:** Intel Xeon Phi 3120A

- Диффузия и лин. алгебра – на OpenMP
- **Векторизация вычислений вручную с помощью AVX-512**
- Раскрутка циклов
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Использование больших страниц памяти

## MIC:

- 512-битная векторизация на MIC необходима для получения высокой производительности, подобно тому как на GPU необходим 32/128-байтовый коалесинг
- Компилятор способен векторизовать простые циклы, однако на сложных циклах – часто консервативен
- Может потребоваться векторизовать вручную по крайней мере часть кода приложения
- На существующих моделях MIC векторные инструкции работают только с выровненными адресами

## GPU/GPU:

- В GPU очень мало векторной арифметики, но есть векторная загрузка/выгрузка данных
- Компилятор старается сам векторизовать код, когда из одного потока происходит обращение к подряд-идущим индексам массива
- Признаком успешной векторизации является присутствие LD.128/ST.128 в дисассемблере `cuobjdump -sass`

# Векторизация

Скалярный код:

```
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];
}
```

Векторизация для MIC:

```
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)))) );
}
```

Векторизация для GPU:

```
template<short V, typename T>
__global__ void kernel(double* __restrict__ y, const double alpha, const double* const __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);
}
```

# Векторизация: результаты на Xeon Phi

- Векторизация влияет на пропускную способность памяти, используемую приложением

версия	пропускная способность памяти
векторная только диффузия	<b>20,86</b> GB/sec
все ядра векторизованы	<b>25.71</b> GB/sec

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

# Векторизация: результаты на GPU для теста *wave13pt*

- Векторизованный код быстрее на 15%
- 2-элементная векторизация улучшает эффективность памяти и уменьшает долю арифметики (меньше индексации?)



Figure: «Наивная» CUDA-реализация

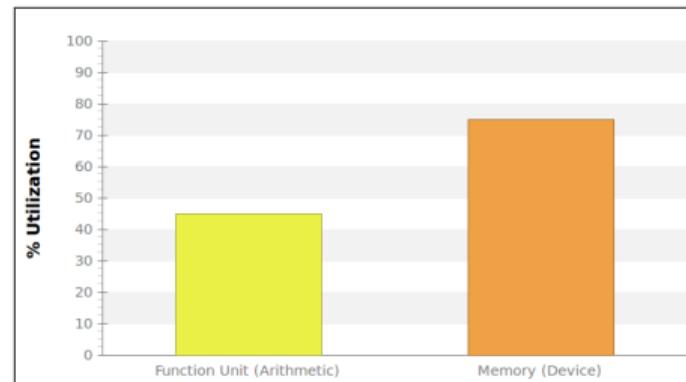


Figure: Векторизованная CUDA-реализация

- Кстати, в этом примере приложение превратилось из compute-bound в memory-bound в результате оптимизации! ☺

# Векторизация: результаты на GPU для теста *wave13pt*

- Векторизованный код быстрее на 15%
- 2-элементная векторизация улучшает эффективность памяти и уменьшает долю арифметики (меньше индексации?)



Figure: «Наивная» CUDA-реализация

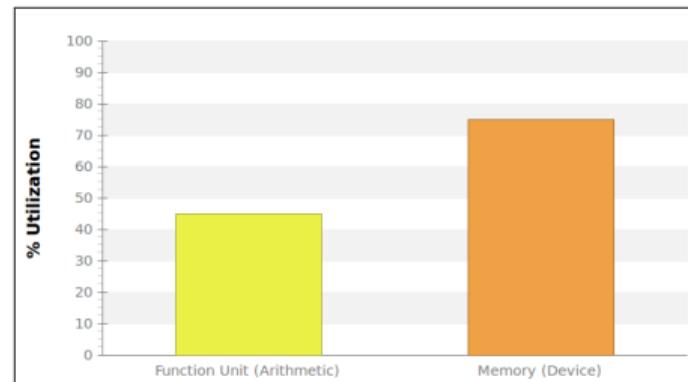


Figure: Векторизованная CUDA-реализация

- Кстати, в этом примере приложение превратилось из compute-bound в memory-bound в результате оптимизации! ☺

# Векторизация: результаты на GPU для теста *wave13pt*

- Пропускная способность памяти выше в векторизованной версии (очень близка к *cuMemcpyDtoD*, кот. даёт 84 Гб/сек на данном устройстве).

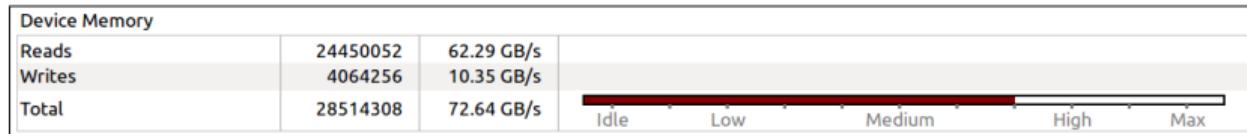


Figure: «Наивная» CUDA-реализация

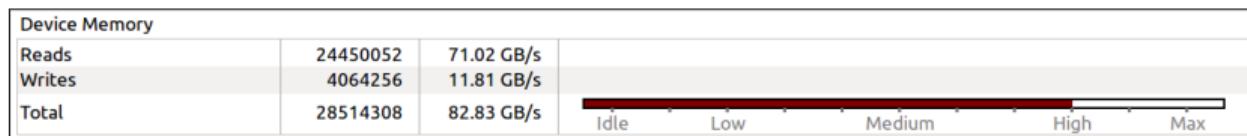


Figure: Векторизованная CUDA-реализация

- На линейной алгебре в Приложении №1 векторизация ничего не даёт

# Выравнивание памяти

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

# Выравнивание памяти

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:

- Паддинг массивов вручную или `cudaArray*`
- Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
__m512d __mm512_load_pd (void const* mem_addr);
__m512d __mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

## GPU/GPU:

- Не поддерживается чтение невыровненных скалярных типов (“Warp Misaligned Address”)
- Выровненные и невыровненные векторные LD/ST поддерживаются прозрачно
- Производительность выше, если каждая транзакция варпа начинается с адреса, выровненного по размеру транзакции:
  - Паддинг массивов вручную или `cudaArray*`
  - Маскирование граничных варпов:

```
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:

- Доступ к скалярным данным возможен с произвольным выравниванием
- Невыровненные векторные LD/ST возможны, но требуется использовать отдельную (и более дорогую) команду для невыровненных адресов (иначе – “Segmentation fault”):

```
_mm512d _mm512_load_pd (void const* mem_addr);
_mm512d _mm512_loadu_pd (void const* mem_addr);
```

- `_mm512_loadu_pd` не поддерживается существующими Xeon Phi (KNL)

# Текстурная память: GPU (`__restrict__`)

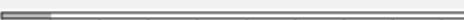
Ключевое слово `__restrict__` позволяет быстро организовать кеширование входных массивов в текстурной памяти:

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

Без `__restrict__`:

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	1325959	84.821 MB/s	
Local Stores	1438647	69.719 MB/s	
Shared Loads	234554352	32.196 GB/s	
Shared Stores	212178170	29.124 GB/s	
Global Loads	1833309331	103.724 GB/s	
Global Stores	524481262	30.829 GB/s	
L1/Shared Total	2807314721	196.027 GB/s	
L2 Cache			
Reads	6053215952	103.86 GB/s	
Writes	1804739214	30.965 GB/s	
Total	7857955166	134.826 GB/s	
Texture Cache			
Reads	276456	4.743 MB/s	
Device Memory			
Reads	5152888875	88.413 GB/s	
Writes	2088264331	35.83 GB/s	
Total	7241153206	124.243 GB/s	
System Memory [ PCIe configuration: Gen2 x16, 5 Gbit/s ]			
Reads	1568	26.903 kB/s	
Writes	202	3.465 kB/s	
Total	1770	30.368 kB/s	

C `__restrict__`:

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	1226172	87.153 MB/s	
Local Stores	1352948	84.938 MB/s	
Shared Loads	234551023	33.34 GB/s	
Shared Stores	198616970	28.232 GB/s	
Global Loads	143521657	10.062 GB/s	
Global Stores	524481262	31.926 GB/s	
L1/Shared Total	1103750832	103.733 GB/s	
L2 Cache			
Reads	5073645870	90.149 GB/s	
Writes	1803794225	32.05 GB/s	
Total	6877440095	122.2 GB/s	
Texture Cache			
Reads	555323056	98.671 GB/s	
Device Memory			
Reads	5043610960	89.616 GB/s	
Writes	2058069967	36.578 GB/s	
Total	7102220627	126.193 GB/s	
System Memory [ PCIe configuration: Gen2 x16, 5 Gbit/s ]			
Reads	137	2.434 kB/s	
Writes	227	4.033 kB/s	
Total	364	6.467 kB/s	

Видно, что текстурная память берёт на себя часть кеширования.

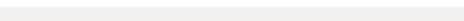
Однако, на производительность приложения №1 это практически не влияет.

# Текстурная память: GPU (`__restrict__`)

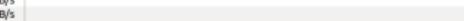
Ключевое слово `__restrict__` позволяет быстро организовать кеширование входных массивов в текстурной памяти:

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

Без `__restrict__`:

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	1325959	84.821 MB/s	
Local Stores	1438647	69.719 MB/s	
Shared Loads	234554352	32.196 GB/s	
Shared Stores	212178170	29.124 GB/s	
Global Loads	1833309331	103.724 GB/s	
Global Stores	524481262	30.829 GB/s	
L1/Shared Total	2807314721	196.027 GB/s	
L2 Cache			
Reads	6053215952	103.86 GB/s	
Writes	1804739214	30.965 GB/s	
Total	7857955166	134.826 GB/s	
Texture Cache			
Reads	276456	4.743 MB/s	
Device Memory			
Reads	5152888875	88.413 GB/s	
Writes	2088246331	35.83 GB/s	
Total	7241153206	124.243 GB/s	
System Memory [ PCIe configuration: Gen2 x16, 5 Gbit/s ]			
Reads	1568	26.903 kB/s	
Writes	202	3.465 kB/s	
Total	1770	30.368 kB/s	

C `__restrict__`:

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	1226172	87.153 MB/s	
Local Stores	1352948	84.938 MB/s	
Shared Loads	234551023	33.34 GB/s	
Shared Stores	198616970	28.232 GB/s	
Global Loads	143521657	10.062 GB/s	
Global Stores	524481262	31.926 GB/s	
L1/Shared Total	1103750832	103.733 GB/s	
L2 Cache			
Reads	5073645870	90.149 GB/s	
Writes	1803794225	32.05 GB/s	
Total	6877440095	122.2 GB/s	
Texture Cache			
Reads	555323056	98.671 GB/s	
Device Memory			
Reads	5043610960	89.616 GB/s	
Writes	2058069967	36.578 GB/s	
Total	7102220627	126.193 GB/s	
System Memory [ PCIe configuration: Gen2 x16, 5 Gbit/s ]			
Reads	137	2.434 kB/s	
Writes	227	4.033 kB/s	
Total	364	6.467 kB/s	

Видно, что текстурная память берёт на себя часть кеширования.

Однако, на производительность приложения №1 это практически не влияет.

# Приложение №1: Реализация



**GPU:** NVIDIA Tesla K20c

- Ядро диффузии на CUDA
- Ядра функций линейной алгебры на CUDA
- Уменьшение синхронизаций (ожидания завершения CUDA-ядер)
- Векторизация LD/ST
- Специализация ядер
- Динамический параллелизм: весь алгоритм как одно GPU-ядро



**GPU:** Gigabyte Radeon HD 7970

- Ядро диффузии на OpenCL
- Ddot и Daxpy – из clAmdBlas, остальное – OpenCL
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Специализация ядер



**MIC:** Intel Xeon Phi 3120A

- Диффузия и лин. алгебра – на OpenMP
- Векторизация вычислений вручную с помощью AVX-512
- Раскрутка циклов
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Использование больших страниц памяти

# Уменьшение эффекта синхронизаций: GPU

- **GPU:** Вызов `cudaDeviceSynchronize` дорог, как на хосте, так и при динамическом параллелизме
- **GPU:** Вызов `clFlush` дорог

```
void gpu:ss_lco... void gpu:diffusi... void gpu:ss_scal... void gpu:ss... void gpu:ss_ax... void gpu:ss_axp... void gp...
```

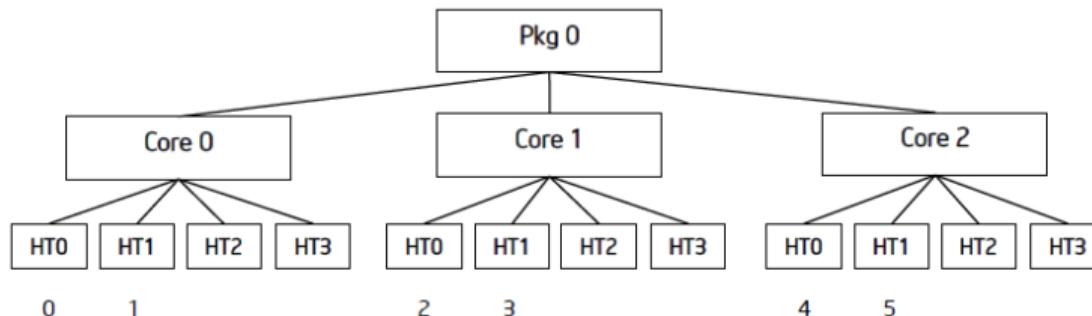
Оптимизация: Синхронизация между ядрами с зависимыми данными происходит неявно  $\Rightarrow$  во многих случаях можно убрать синхронизацию, тем самым уменьшив простои:

```
void gpu:ss_lcomb_k... void gpu:diffusion_in... void gpu:ss_scaled_di... void gpu:ss_dot ... void gpu:ss_axpy_ke... void gpu:ss_axpy_ker... void gp...
```

(Время ядер при этом не меняется, например, `ss_axpy`:  $179.653 \mu s \rightarrow 178.021 \mu s.$ )

# Уменьшение эффекта синхронизаций: MIC

- MIC: Runtime-библиотека OpenMP синхронизирует потоки, и может делать это долго, если нагрузки разбалансированы:
  - Отсутствие настройки affinity  $\Rightarrow$  поток может свободно мигрировать между ядрами. На CPU это малозаметно, т.к. помогает кеш L3. На MIC L3-кеша нет  $\Rightarrow$  эффективность потока может снизиться, если он мигрирует далеко от своего кеша. Настройка KMP\_AFFINITY=granularity=fine,balanced привязывает логический поток к физическим потокам в ядрах:



- На Xeon Phi работает ОС Linux. Имеет смысл выделить под неё отдельное ядро с помощью KMP\_PLACE\_THREADS=56c,4t

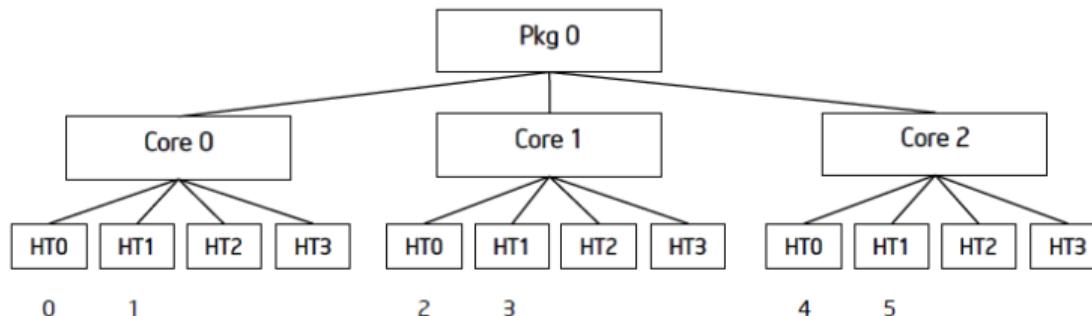
```
$ micnativeloadex ./main -a "512 512 400 0.01"  
simulation took 31.7525 seconds
```



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

# Уменьшение эффекта синхронизаций: MIC

- **MIC**: Runtime-библиотека OpenMP синхронизирует потоки, и может делать это долго, если нагрузки разбалансированы:
  - Отсутствие настройки `affinity` ⇒ поток может свободно мигрировать между ядрами. На CPU это малозаметно, т.к. помогает кеш L3. На MIC L3-кеша нет ⇒ эффективность потока может снизиться, если он мигрирует далеко от своего кеша. Настройка `KMP_AFFINITY=granularity=fine,balanced` привязывает логический поток к физическим потокам в ядрах:



- На Xeon Phi работает ОС Linux. Имеет смысл выделить под неё отдельное ядро с помощью `KMP_PLACE_THREADS=56c,4t`

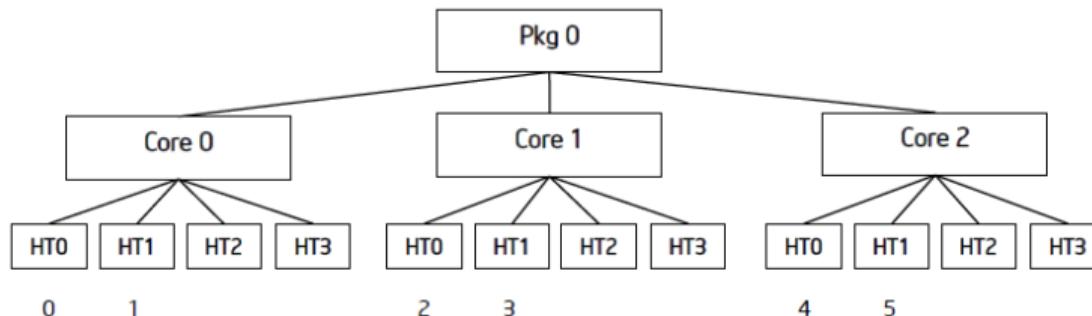
```
$ micnativeloadex ./main -a "512 512 400 0.01"  
simulation took 31.7525 seconds
```



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

# Уменьшение эффекта синхронизаций: MIC

- **MIC**: Runtime-библиотека OpenMP синхронизирует потоки, и может делать это долго, если нагрузки разбалансированы:
  - Отсутствие настройки affinity  $\Rightarrow$  поток может свободно мигрировать между ядрами. На CPU это малозаметно, т.к. помогает кеш L3. На MIC L3-кеша нет  $\Rightarrow$  эффективность потока может снизиться, если он мигрирует далеко от своего кеша. Настройка KMP\_AFFINITY=granularity=fine,balanced привязывает логический поток к физическим потокам в ядрах:



- На Xeon Phi работает ОС Linux. Имеет смысл выделить под неё отдельное ядро с помощью KMP\_PLACE\_THREADS=56c,4t

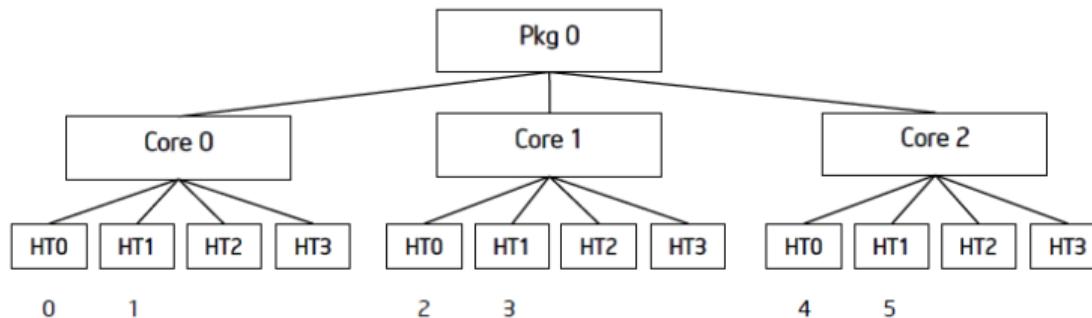
```
$ micnativeloadex ./main -a "512 512 400 0.01"  
simulation took 31.7525 seconds
```



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

# Уменьшение эффекта синхронизаций: MIC

- **MIC**: Runtime-библиотека OpenMP синхронизирует потоки, и может делать это долго, если нагрузки разбалансированы:
  - Отсутствие настройки affinity  $\Rightarrow$  поток может свободно мигрировать между ядрами. На CPU это малозаметно, т.к. помогает кеш L3. На MIC L3-кеша нет  $\Rightarrow$  эффективность потока может снизиться, если он мигрирует далеко от своего кеша. Настройка KMP\_AFFINITY=granularity=fine,balanced привязывает логический поток к физическим потокам в ядрах:



- На Xeon Phi работает ОС Linux. Имеет смысл выделить под неё отдельное ядро с помощью  
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
```

# Приложение №1: Реализация



**GPU:** NVIDIA Tesla K20c

- Ядро диффузии на CUDA
- Ядра функций линейной алгебры на CUDA
- Уменьшение синхронизаций (ожидания завершения CUDA-ядер)
- Векторизация LD/ST
- **Специализация ядер**
- Динамический параллелизм: весь алгоритм как одно GPU-ядро



**GPU:** Gigabyte Radeon HD 7970

- Ядро диффузии на OpenCL
- Ddot и Daxpy – из clAmdBlas, остальное – OpenCL
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- **Специализация ядер**



**MIC:** Intel Xeon Phi 3120A

- Диффузия и лин. алгебра – на OpenMP
- Векторизация вычислений вручную с помощью AVX-512
- Раскрутка циклов
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Использование больших страниц памяти

# Специализация ядер: GPU

```
__global__ 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 + blockIdx.x * blockDim.x, y = threadIdx.y + blockIdx.y * blockDim.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;
    }
}
```

# Специализация ядер: GPU

Приём специализации состоит в разделении ядра с условием на индекс на два и запуске их друг за другом:

```
__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];
}
```

# Специализация ядер: GPU

Приём специализации состоит в разделении ядра с условием на индекс на два и запуске их друг за другом:

```
__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 = (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;
}
```

Результат: время работы вместо 0.826112 сек стало 0.728256 сек.

# Приложение №1: Реализация



**GPU:** NVIDIA Tesla K20c

- Ядро диффузии на CUDA
- Ядра функций линейной алгебры на CUDA
- Уменьшение синхронизаций (ожидания завершения CUDA-ядер)
- Векторизация LD/ST
- Специализация ядер
- Динамический параллелизм: весь алгоритм как одно GPU-ядро



**GPU:** Gigabyte Radeon HD 7970

- Ядро диффузии на OpenCL
- Ddot и Daxpy – из clAmdBlas, остальное – OpenCL
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Специализация ядер



**MIC:** Intel Xeon Phi 3120A

- Диффузия и лин. алгебра – на OpenMP
- Векторизация вычислений вручную с помощью AVX-512
- Раскрутка циклов
- Уменьшение синхронизаций (ожидания завершения OpenMP-циклов)
- Использование больших страниц памяти

# Использование больших страниц памяти

- Большие страницы (huge pages) решают проблему медленного выделения памяти на [MIC](#)
- По умолчанию в Linux используются страницы памяти размером 4 Кбайт; большая страница имеет размер 2 Мбайт

## Метод:

- Переопределить `malloc/free` так, чтобы они использовали `mmap` с флагом `MAP_HUGETLB`

## Результат:

- Цикл с простым копированием массива через `memcpuy` ускорился в 2 раза

## [Статья на сайте Intel](#)

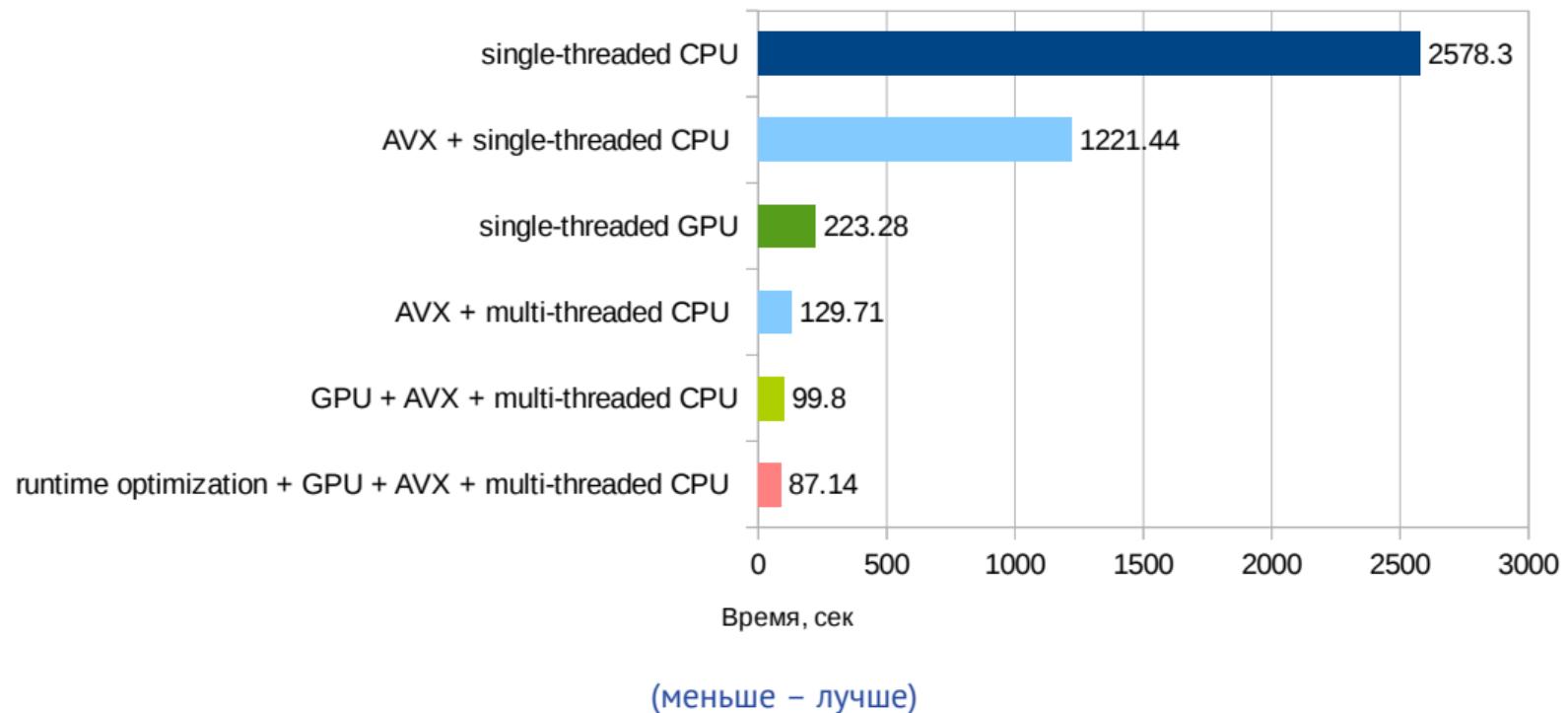
### Adaptive Sparse Grids Solver for High-Dimensional Dynamic Models

(Johannes Brumm and Simon Scheidegger @ University of Zurich)

- Решается задача оптимизации в модели экономических процессов
- По мере решения требуется вычислять значения целевой функции в заданных точках
- Вычисление значений функции занимает 90% времени

- 1 Упрощение выражений для исключения делений
- 2 Исключение дублирующий вычислений, т.ч. не повышались byte на FLOPS, исключение if-ов
- 3 Распараллеливание на Thrust с помощью `transform_reduce`
- 4 Исключение избыточных `cudaMalloc/cudaFree` из Thrust-реализации
- 5 Runtime-оптимизация: компиляция ядер под конкретный размер вектора, т.ч. элементы вектора передаются как скалярные аргументы
  - Выигрыш 15%, но нужно компилировать ядро во время исполнения и подгружать из DLL
  - Использует диск  $\Rightarrow$  может быть медленно на ФС кластера, нужно писать singleton для MPI/threads
- 6 Многопоточность на CPU с помощью Intel TBB + 1 поток на GPU, нагрузка перераспределяется автоматически за счёт work stealing

## Приложение №2:



- Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- Векторизация на MIC и коалесинг на GPU/GPU требуют сопоставимых усилий по переписыванию кода
- Реальная пропускная способность памяти на MIC пока получается самой низкой
- Профилировщики для GPU и GPU нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- 3 Векторизация на MIC и коалесинг на GPU/GPU требуют сопоставимых усилий по переписыванию кода
- 4 Реальная пропускная способность памяти на MIC пока получается самой низкой
- 5 Профилировщики для GPU и GPU нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- 3 Векторизация на MIC и коалесинг на GPU/GPU требуют сопоставимых усилий по переписыванию кода
- 4 Реальная пропускная способность памяти на MIC пока получается самой низкой
- 5 Профилировщики для GPU и GPU нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- 3 Векторизация на MIC и коалесинг на GPU/GPU требуют сопоставимых усилий по переписыванию кода
- 4 Реальная пропускная способность памяти на MIC пока получается самой низкой
- 5 Профилировщики для GPU и GPU нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
  - 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
  - 3 Векторизация на **MIC** и коалесинг на **GPU/GPU** требуют сопоставимых усилий по переписыванию кода
  - 4 Реальная пропускная способность памяти на **MIC** пока получается самой низкой
-  Профилировщики для **GPU** и **GPU** нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- 3 Векторизация на **MIC** и коалесинг на **GPU/GPU** требуют сопоставимых усилий по переписыванию кода
- 4 Реальная пропускная способность памяти на **MIC** пока получается самой низкой
- 5 Профилировщики для **GPU** и **GPU** нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).

- 1 Выбирать тип ускорителя лучше всего для известного целевого приложения, после проведения анализа и оптимизаций на тестовой системе (которую мы Вам можем предоставить!)
- 2 Ориентируясь на ускоритель, имеет смысл разрабатывать гибридное решение, как в Приложении №2
- 3 Векторизация на **MIC** и коалесинг на **GPU/GPU** требуют сопоставимых усилий по переписыванию кода
- 4 Реальная пропускная способность памяти на **MIC** пока получается самой низкой
- 5 Профилировщики для **GPU** и **GPU** нуждаются в доработке

Для получения слайдов и примеров программ к данной презентации, до 16 июля отправьте 10-15 слайдов о Вашем проекте на любом ускорителе на адрес: [contact@parallel-computing.pro](mailto:contact@parallel-computing.pro).