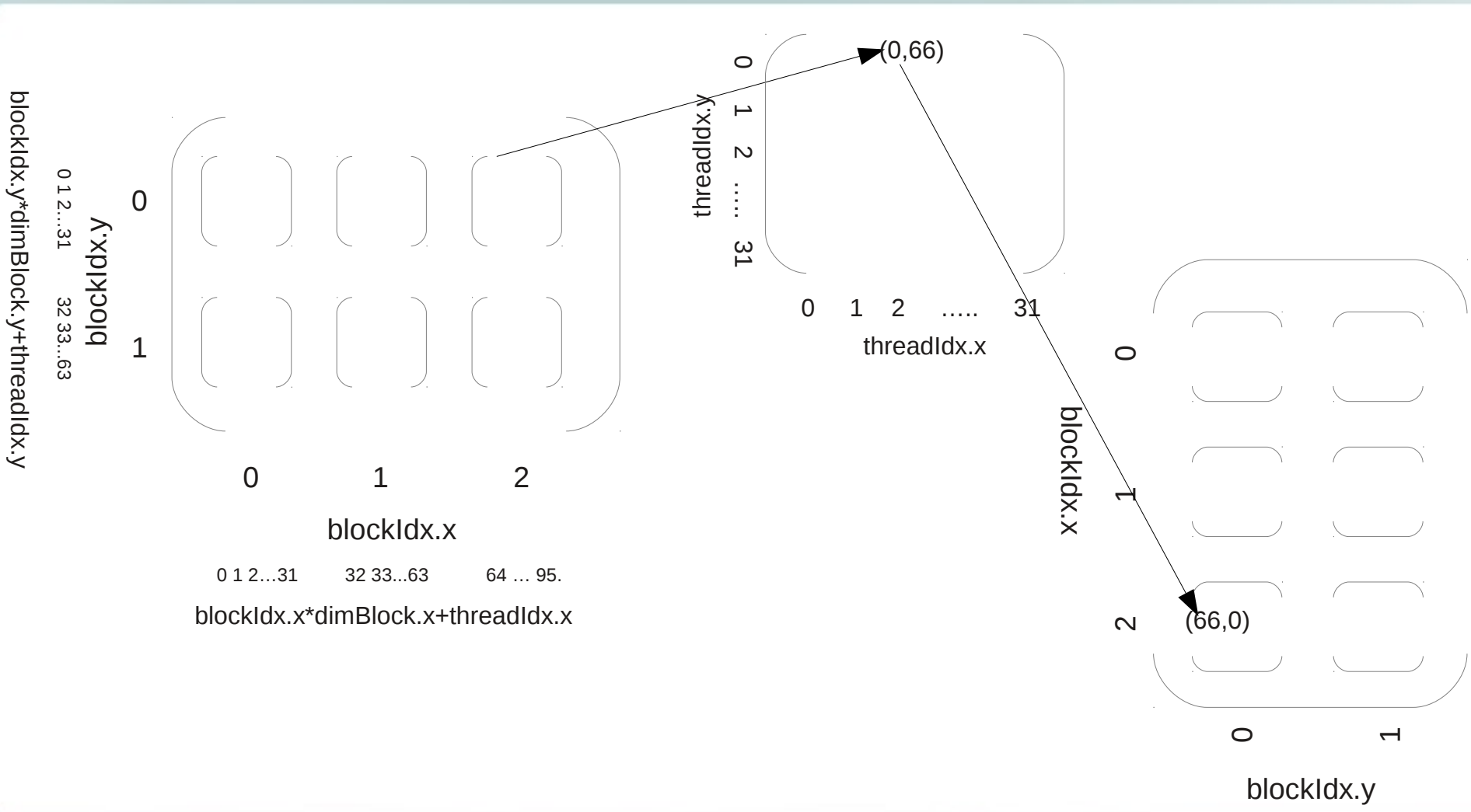
Лекция 3

- разбор алгоритмов распараллеливания транспонироания матриц – подробнее о *coalescing*'е и *shared memory*;
- алгоритмы распараллеливания суммирования, реализация процедуры reduce вручную;
- использование библиотеки ***thrust***, суммирование с использованием алгоритмов этой библиотеки.

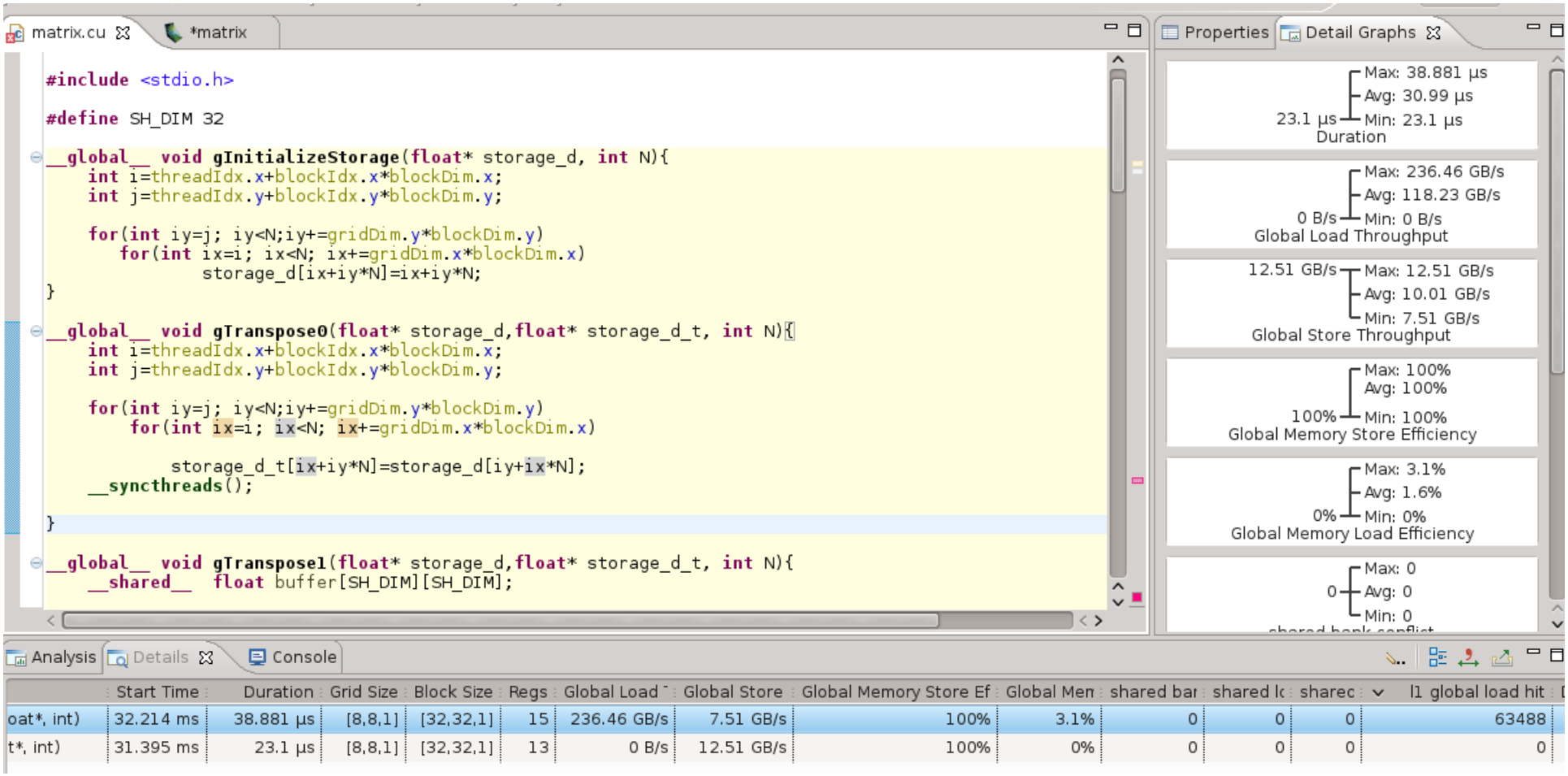
Простое транспонирование

```
#include <stdio.h>

#define SH_DIM 32

__global__ void gInitializeStorage(float* storage_d, int N){
        int i=threadIdx.x+blockIdx.x*blockDim.x;
        int j=threadIdx.y+blockIdx.y*blockDim.y;

        for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)      //цикл позволяет выбирать произвольное значение N, меньше
          for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)  //количества потоков в блоке, и больше общего количества потоков
                        storage_d[ix+iy*N]=ix+iy*N;
}

__global__ void gTranspose0(float* storage_d,float* storage_d_t, int N){
        int i=threadIdx.x+blockIdx.x*blockDim.x;
        int j=threadIdx.y+blockIdx.y*blockDim.y;

        for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)
                for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)

                        storage_d_t[iy+ix*N]=storage_d[ix+iy*N];
        __syncthreads();

}
```

Простое транспонирование (схема)



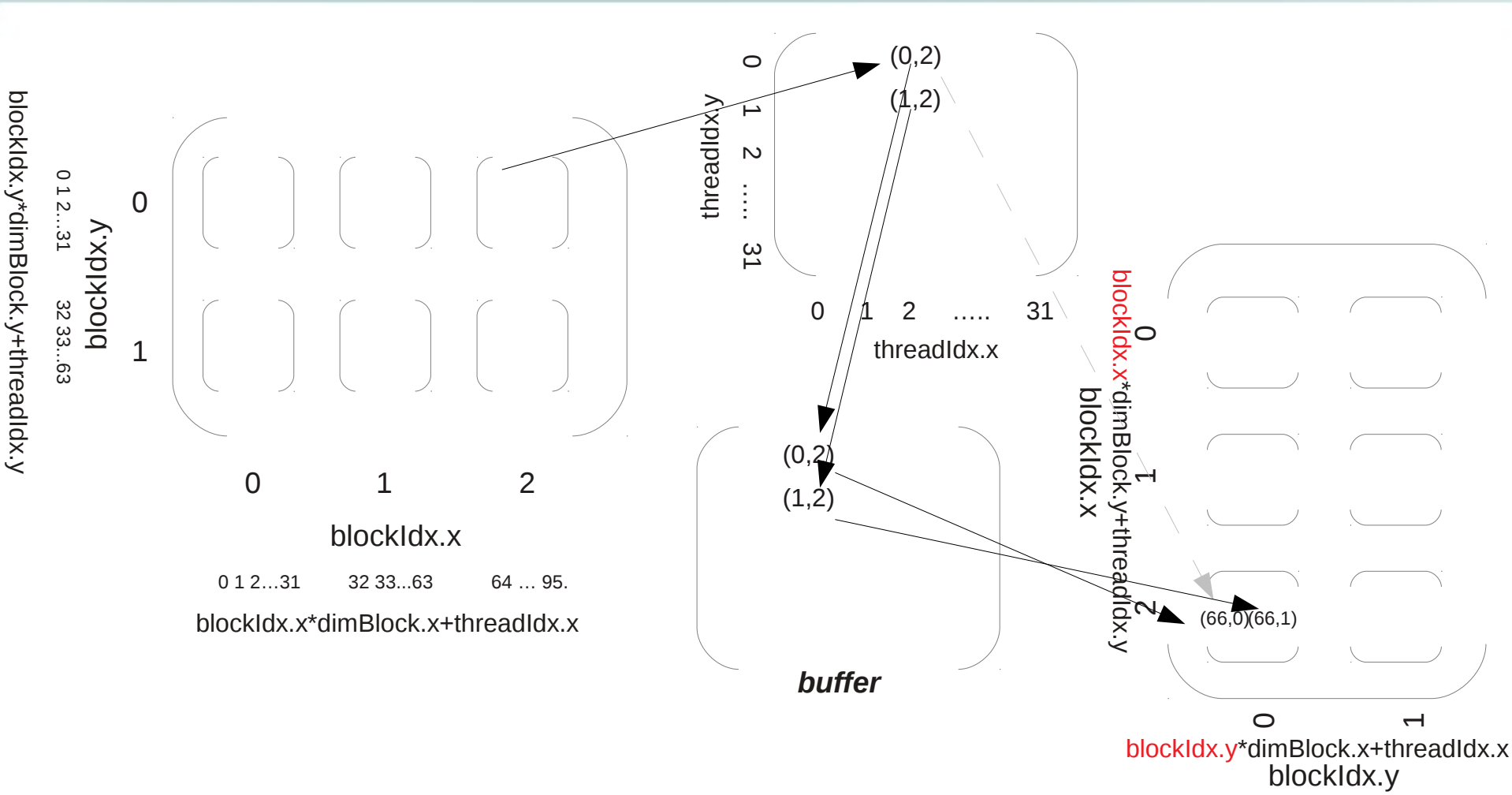
Простое транспонирование (результат профилирования)



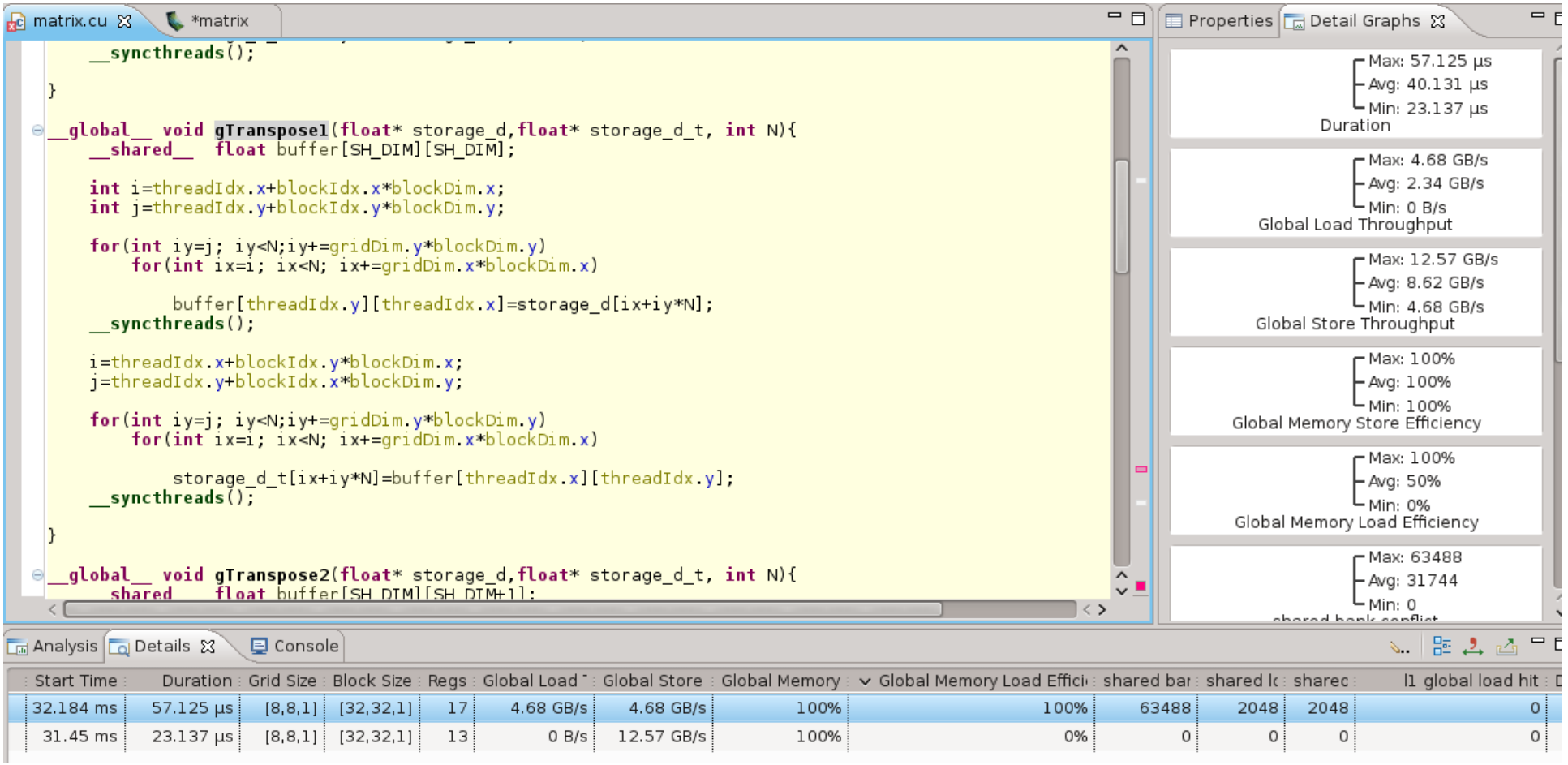
Использование shared memory с конфликтом банков

```
__global__ void gTranspose1(float* storage_d,float* storage_d_t, int N){
	__shared__  float buffer[SH_DIM][SH_DIM];

	int i=threadIdx.x+blockIdx.x*blockDim.x;
	int j=threadIdx.y+blockIdx.y*blockDim.y;

	for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)
		for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)

			buffer[threadIdx.y][threadIdx.x]=storage_d[ix+iy*N];
	__syncthreads();

	i=threadIdx.x+blockIdx.y*blockDim.x;
	j=threadIdx.y+blockIdx.x*blockDim.y;

	for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)
		for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)

			storage_d_t[ix+iy*N]=buffer[threadIdx.x][threadIdx.y];
	__syncthreads();

}
```

Использование shared memory с конфликтом банков (схема)



Использование shared memory с конфликтами банков (результат профилирования)



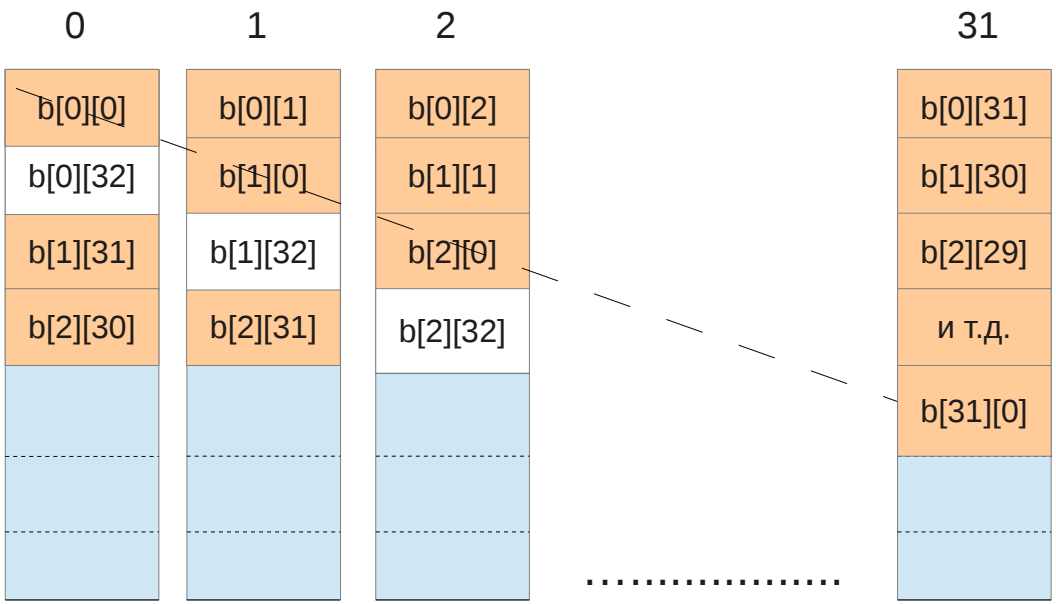
Использование shared memory с разрешением конфликтов банков

```
__global__ void gTranspose2(float* storage_d,float* storage_d_t, int N){
	__shared__  float buffer[SH_DIM][SH_DIM+1];

	int i=threadIdx.x+blockIdx.x*blockDim.x;
	int j=threadIdx.y+blockIdx.y*blockDim.y;

	for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)
		for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)

			buffer[threadIdx.y][threadIdx.x]=storage_d[ix+iy*N];
	__syncthreads();

	i=threadIdx.x+blockIdx.y*blockDim.x;
	j=threadIdx.y+blockIdx.x*blockDim.y;

	for(int iy=j; iy<N;iy+=gridDim.y*blockDim.y)
		for(int ix=i; ix<N; ix+=gridDim.x*blockDim.x)

			storage_d_t[ix+iy*N]=buffer[threadIdx.x][threadIdx.y];
	__syncthreads();

}
```

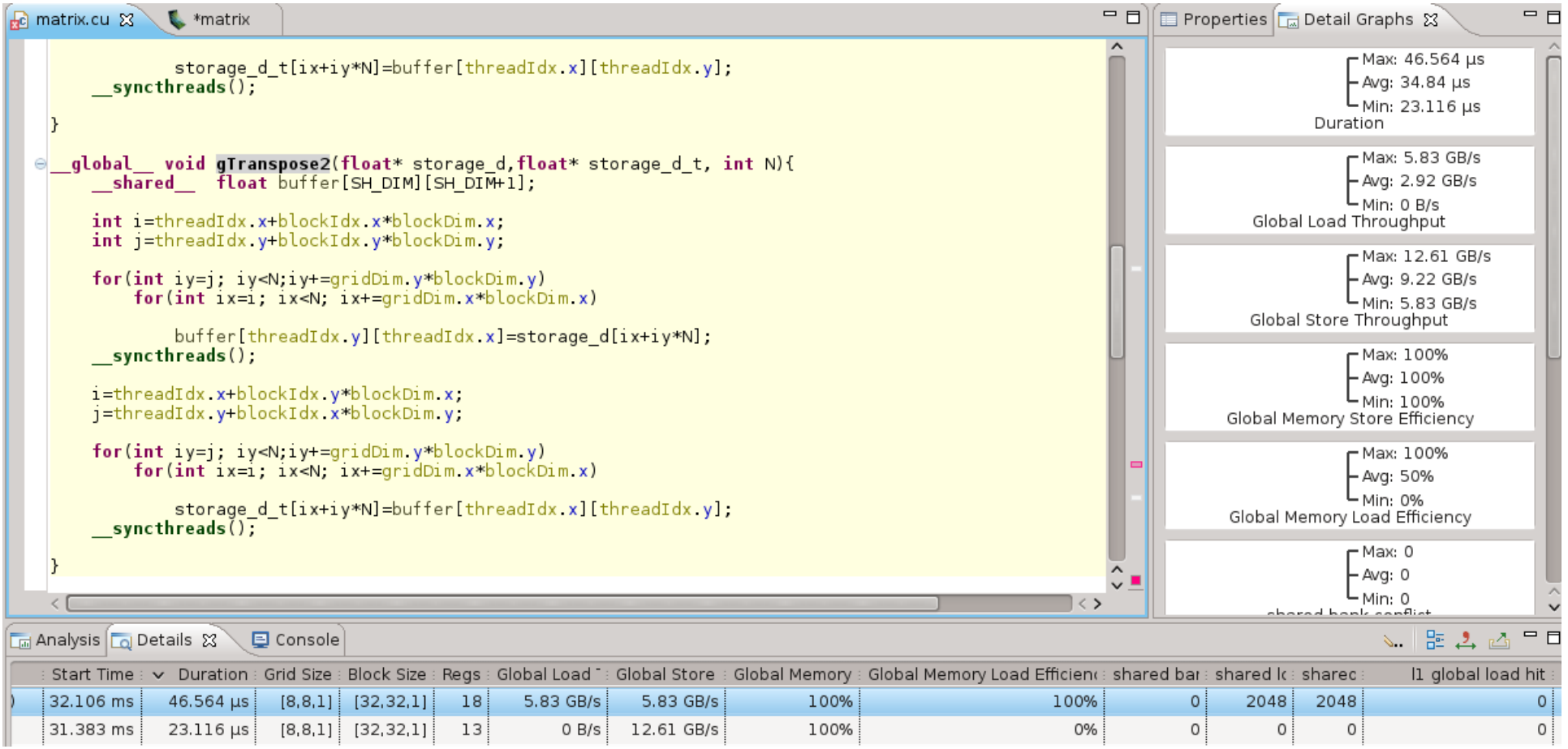
Как избежать конфликта банков разделяемой памяти

__shared__ float buffer[SH_DIM][SH_DIM+1];

Размещение массива buffer в разделяемой памяти (shared memory):



Использование shared memory с разрешением конфликтов банков (результат прфилирования)



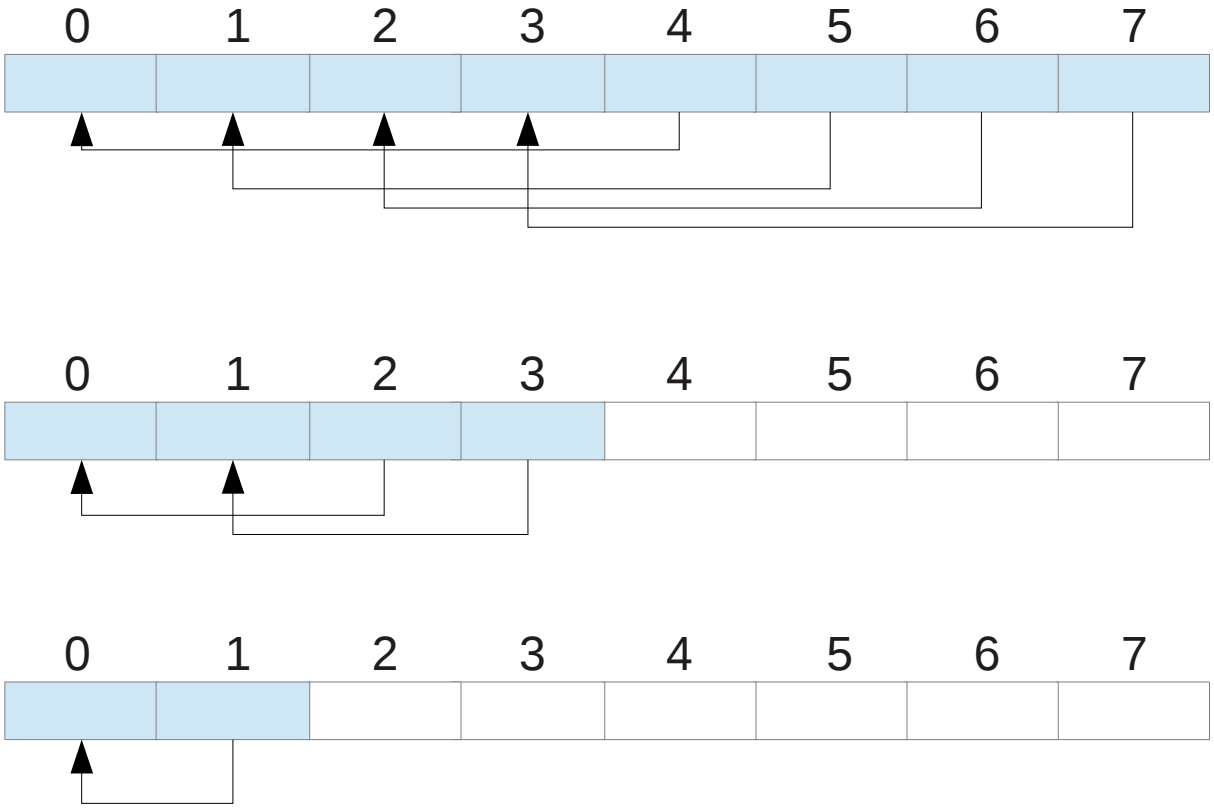
Конфигурация нитей и запуск ядра

```
int main(int argc, char* argv[]){
        if(argc<2){
                fprintf(stderr, "USAGE: matrix  <dimension of matrix>\n");
                return -1;
        }
        int N=atoi(argv[1]);

        const int max_size=1024;
        int size=N/32+(N%32>0);
        int dim_of_blocks=(size>max_size)?max_size:size;
        int dim_of_threads=32;
        float *storage_d, *storage_d_t, *storage_h;

        cudaMalloc((void**)&storage_d, N*N*sizeof(float));
        cudaMalloc((void**)&storage_d_t, N*N*sizeof(float));
        storage_h=(float*)calloc(N*N, sizeof(float));

        gInitializeStorage<<<dim3(dim_of_blocks, dim_of_blocks),dim3(dim_of_threads,dim_of_threads)>>>(storage_d,N);
        cudaThreadSynchronize();

        gTranspose2<<<dim3(dim_of_blocks,     dim_of_blocks),dim3(dim_of_threads,dim_of_threads)>>>(storage_d,storage_d_t,N);
        cudaThreadSynchronize();

        cudaMemcpy(storage_h,  storage_d_t, N*N*sizeof(float), cudaMemcpyDeviceToHost);
        Output(storage_h, N);

        cudaFree(storage_d);
        cudaFree(storage_d_t);
        free(storage_h);
       return 0;
}
```

Время выполнения ядра с разными архитектурами CUDA

-arch=sm_12	-arch=sm_21
gTranspose0 took 0.036864 gTranspose0 took 0.038016	gTranspose0 took 0.033664 gTranspose0 took 0.033152
gTranspose1 took 0.039488 gTranspose1 took 0.040128	gTranspose1 took 0.03616 gTranspose1 took 0.03632
gTranspose2 took 0.02752 gTranspose2 took 0.027424	gTranspose2 took 0.024672 gTranspose2 took 0.024544 (N=256)

Редукция



Использование редукции при вычислении скалярного произведения

```
#define REAL float
#define SHMEM_SIZE  512
#include <stdio.h>
__global__ void gScalarProduct(REAL* v1, REAL* v2, int N, REAL* S){
    __shared__ float s[SHMEM_SIZE];

    int i=threadIdx.x;//+blockIdx.x*blockDim.x;

    int i_center;

     s[i]=v1[threadIdx.x+blockIdx.x*blockDim.x]*v2[threadIdx.x+blockIdx.x*blockDim.x];
     __syncthreads();

     i_center=blockDim.x/2;

     while(i_center!=0){
        if(i<i_center)
        s[i]+=s[i+i_center];
        __syncthreads();
        i_center/=2;
    }

    if(threadIdx.x==0)
        atomicAdd(S,s[0]);
}
```

Использование редукции при вычислении скалярного произведения (продолжение)

```
__global__ void gInit(REAL* v){
    v[threadIdx.x+blockIdx.x*blockDim.x]=0.1F*(threadIdx.x+blockIdx.x*blockDim.x);
 }

int main(int argc, char* argv[]){
    int num_of_blocks=atoi(argv[1]);
    int threads_per_block=atoi(argv[2]);

    int N=threads_per_block*num_of_blocks;

    REAL *v1, *v2;
    REAL* pS_d;
    REAL S_h=0;

    cudaMalloc((void**)&v1, N*sizeof(REAL));
    cudaMalloc((void**)&v2, N*sizeof(REAL));
    cudaMalloc((void**)&pS_d, sizeof(REAL));

    gInit<<<num_of_blocks, threads_per_block>>>(v1);
    cudaThreadSynchronize();

    gInit<<<num_of_blocks,threads_per_block >>>(v2);
    cudaThreadSynchronize();

    cudaMemcpy(pS_d,&S_h, sizeof(REAL), cudaMemcpyHostToDevice);
```

Использование редукции при вычислении скалярного произведения (окончание)

```
    cudaEvent_t  start,stop;
    float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start,0);

    gScalarProduct<<<num_of_blocks, threads_per_block>>>(v1, v2, N, pS_d);
    cudaThreadSynchronize();

    cudaMemcpy(&S_h, pS_d,sizeof(REAL), cudaMemcpyDeviceToHost);
    printf("Scalar product is equal to %g\n", S_h);

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
fprintf(stderr,"gScalarProduct took: %g\n",elapsedTime);
cudaEventDestroy(start);
cudaEventDestroy(stop);

    cudaFree(v1);
    cudaFree(v2);
    cudaFree(pS_d);
     return 0;
}
```

Использование средств библиотеки ***thrust*** при вычислении скалярного произведения

```
#include <thrust/device_vector.h>
#include <thrust/inner_product.h>
#include <stdio.h>
#define REAL float

__global__ void gInit(REAL* v){
     v[threadIdx.x+blockIdx.x*blockDim.x]=0.1F*(threadIdx.x+blockIdx.x*blockDim.x);
 }
int main(int argc, char* argv[]){
    REAL *v1, *v2;
    int num_of_blocks=atoi(argv[1]);
    int threads_per_block=atoi(argv[2]);
    int N=threads_per_block*num_of_blocks;

    cudaMalloc((void**)&v1, N*sizeof(REAL));
    cudaMalloc((void**)&v2, N*sizeof(REAL));

    gInit<<<num_of_blocks, threads_per_block>>>(v1);
    cudaThreadSynchronize();

    gInit<<<num_of_blocks,threads_per_block >>>(v2);
    cudaThreadSynchronize();
```

Использование средств библиотеки ***thrust*** при вычислении скалярного произведения (окончание)

```
  thrust::device_ptr<float> v1_ptr = thrust::device_pointer_cast(v1);
  thrust::device_ptr<float> v2_ptr = thrust::device_pointer_cast(v2);

cudaEvent_t  start,stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

  REAL s=thrust::inner_product(v1_ptr, v1_ptr+N, v2_ptr,0.0);
  fprintf(stderr,"Scalar Product (thrust) s =%g\n",s);

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
fprintf(stderr,"Thrust Reduce: %g\n",elapsedTime);
cudaEventDestroy(start);
cudaEventDestroy(stop);
 return 0;
}
```