

# Can the Performance of GPGPU Really Beat CPU in Evolutionary Design Task?

**Václav ŠIMEK**   **Karel SLANÝ**   **Zdeněk VAŠÍČEK**

[simekv@fit.vutbr.cz](mailto:simekv@fit.vutbr.cz)

[slany@fit.vutbr.cz](mailto:slany@fit.vutbr.cz)

[vasicek@fit.vutbr.cz](mailto:vasicek@fit.vutbr.cz)

Brno University of Technology, Faculty of Information Technology,  
Božetěchova 2, 612 66 Brno, Czech Republic

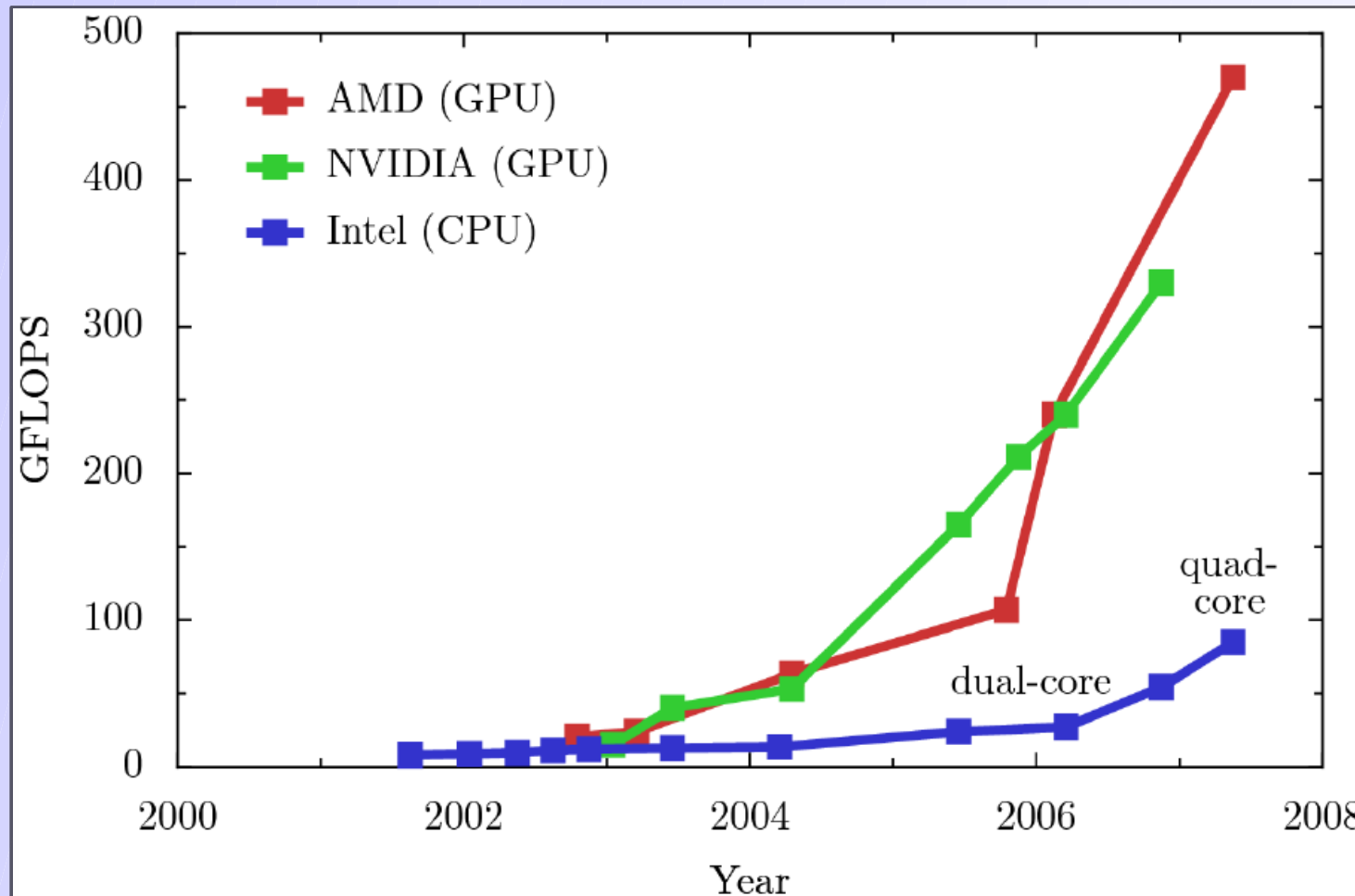


- Recent trends and motivation
- Architecture behind GPGPU
- Practical aspects of using CUDA
- Comments on potential issues
- Acceleration of CGP
- Review and conclusions
- Conclusions

# Recent trends



- Technology improvements brings the continuous growth of processing power.



- Quick and efficient processing of large-scale data sets may easily go beyond the limits of available CPUs.
- The obvious answer to such drawbacks lies in the adoption of dedicated piece of hardware.
  - *multicore CPUs or Cell-based architectures*
  - *designing with FPGA*
  - ***exploitation of commodity GPU chips***
- The speed-up typically reffers to sophisticated design of parallel processing units, instead of higher frequency.
- It's necessary to consider several factors before making the final choice.

# Why to select GPU platform?



- **GPUs are fast...**

- 3.0GHz Intel Core2 Quad:
  - ◆ Computation: 20 GFLOPS
  - ◆ Memory bandwidth: 55 GB/s
- NVIDIA GeForce 8800 GTX:
  - ◆ Computation: 330 GFLOPS
  - ◆ Memory bandwidth: 96 GB/s



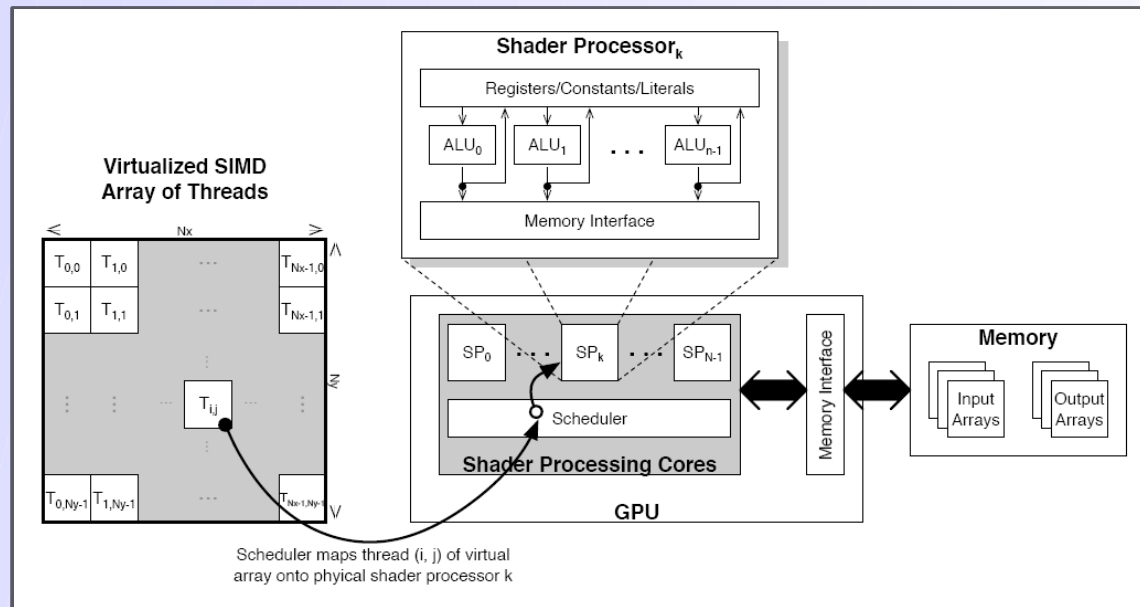
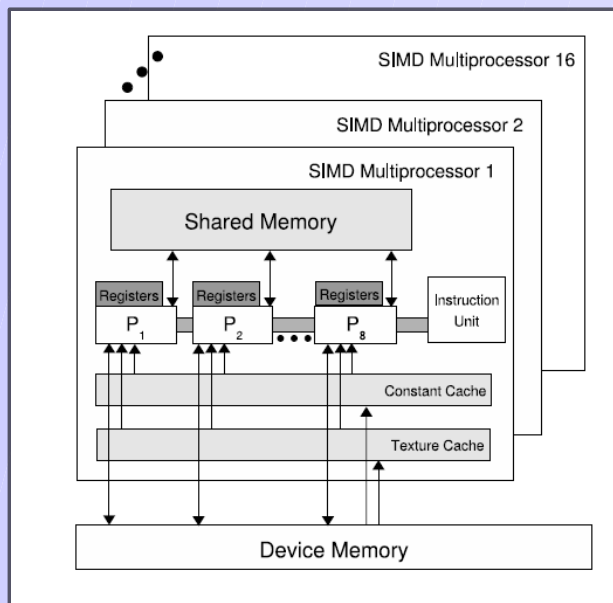
- **Arithmetic intensity...**

- Special nature of GPU makes it possible to devote more transistors to “real” computation.

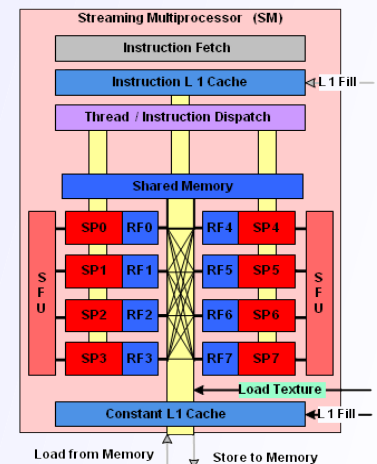
- **Ease of use...**

- Unusual programming model renders the common GPUs a bit clumsy with regard to general tasks
- Unified framework like CUDA or RapidMind changes the tides!

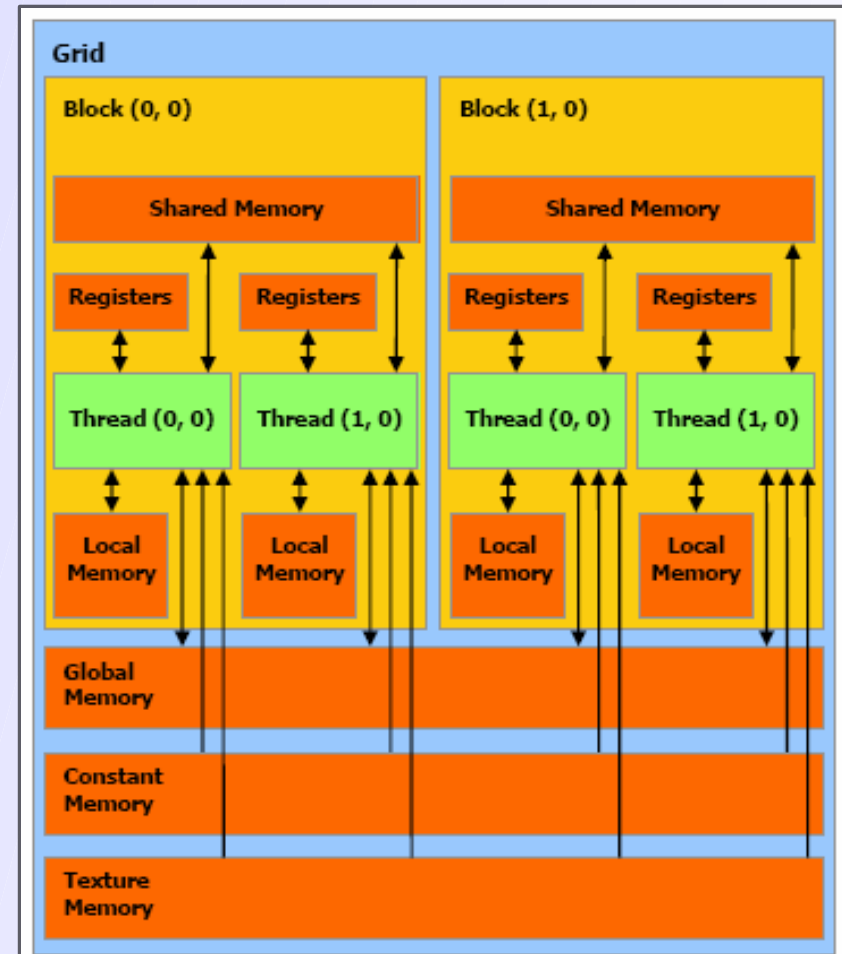
# Overview of nVidia GPU



- GPU is massively parallel architecture
  - there's a large amount of arithmetic capability
  - features specifically designed for computation
- GPU operating characteristics
  - 675MHz for stream multiprocessors (SM)
  - 1350MHz for each individual element within SM

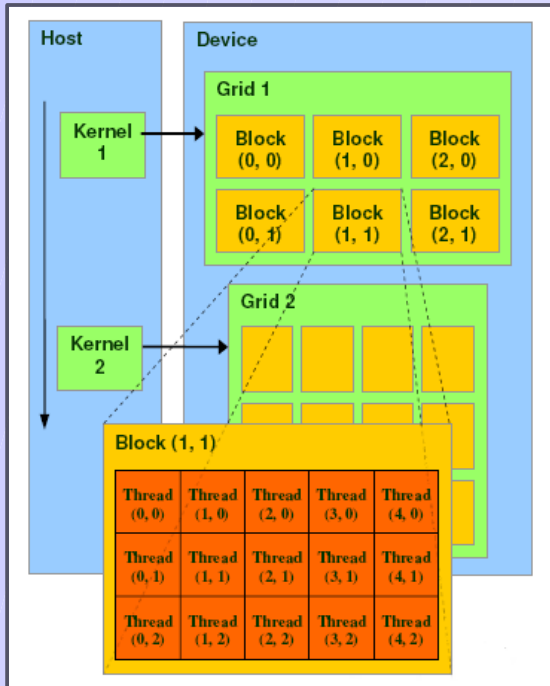


- Cached texture and constant memory
- Unbuffered global memory (400 – 600 cycles latency), accessible from host via library call
- Shared memory (16K each block, low latency 1 cycle)
- Local memory – unshared variables for each thread





- **Resources in GPU can be efficiently used with program partitioned into several threads**
  - interaction between threads through shared on-chip memory
  - memory access management on behalf of the application
- **Parallel **kernels** run a single program in many threads on the device**
  - same code applied to number of processing threads
- **Fully general load-store model**
  - read-write to any location
  - possibility to create and use data pointers
- **Any data type is allowable**
  - available memory looks like a linear sequence of bytes



- Execution of kernels as a grid of threads
  - simplified computation of address in each thread
  - threads synchronization through barrier
- Partition data within well-sized blocks
  - small enough to be placed in shared memory
  - each data partition can be assigned to a thread block
- Several performance benefits
  - it's desirable to keep processors busy
  - working in shared memory hides the access latency

# An example of language constructs



- User application code is written in standard C language with only minor extensions.
- **Function qualifiers:**

```
__global__ void MyKernel() { }  
__device__ float MyDeviceFunc() { }
```

- **Variable qualifiers:**

```
__constant__ float MyConstantArray[32];  
__shared__ float MySharedArray[32];
```

- **Execution configuration:**

```
dim3 dimGrid(100, 50); // 5000 thread blocks  
dim3 dimBlock(4, 8, 8); // 256 threads per block  
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

- **Built-in variables:**

```
dim3 gridDim; // Grid dimension  
dim3 blockDim; // Block dimension  
dim3 blockIdx; // Block index  
dim3 threadIdx; // Thread index
```

- computation of vector sum  $C = A + B$
- each thread performs one pair-wise addition

```
__global__ void test(int * idata1, int * idata2, int * odata) {
    odata[threadIdx.x] = idata1[threadIdx.x] + idata2[threadIdx.x];
    __syncthreads();
}

int main(void) {

    // allocate host and device memory, ...
    // copy host memory to device
    CUDA_SAFE_CALL( cudaMemcpy( d_idata1, arr1i, mem_size, cudaMemcpyHostToDevice) );
    CUDA_SAFE_CALL( cudaMemcpy( d_idata2, arr2i, mem_size, cudaMemcpyHostToDevice) );

    dim3 grid(1, 1, 1); dim3 threads( 8, 1, 1);
    test<<< grid, threads >>>(d_idata1, d_idata2, d_odata);
    cudaThreadSynchronize();

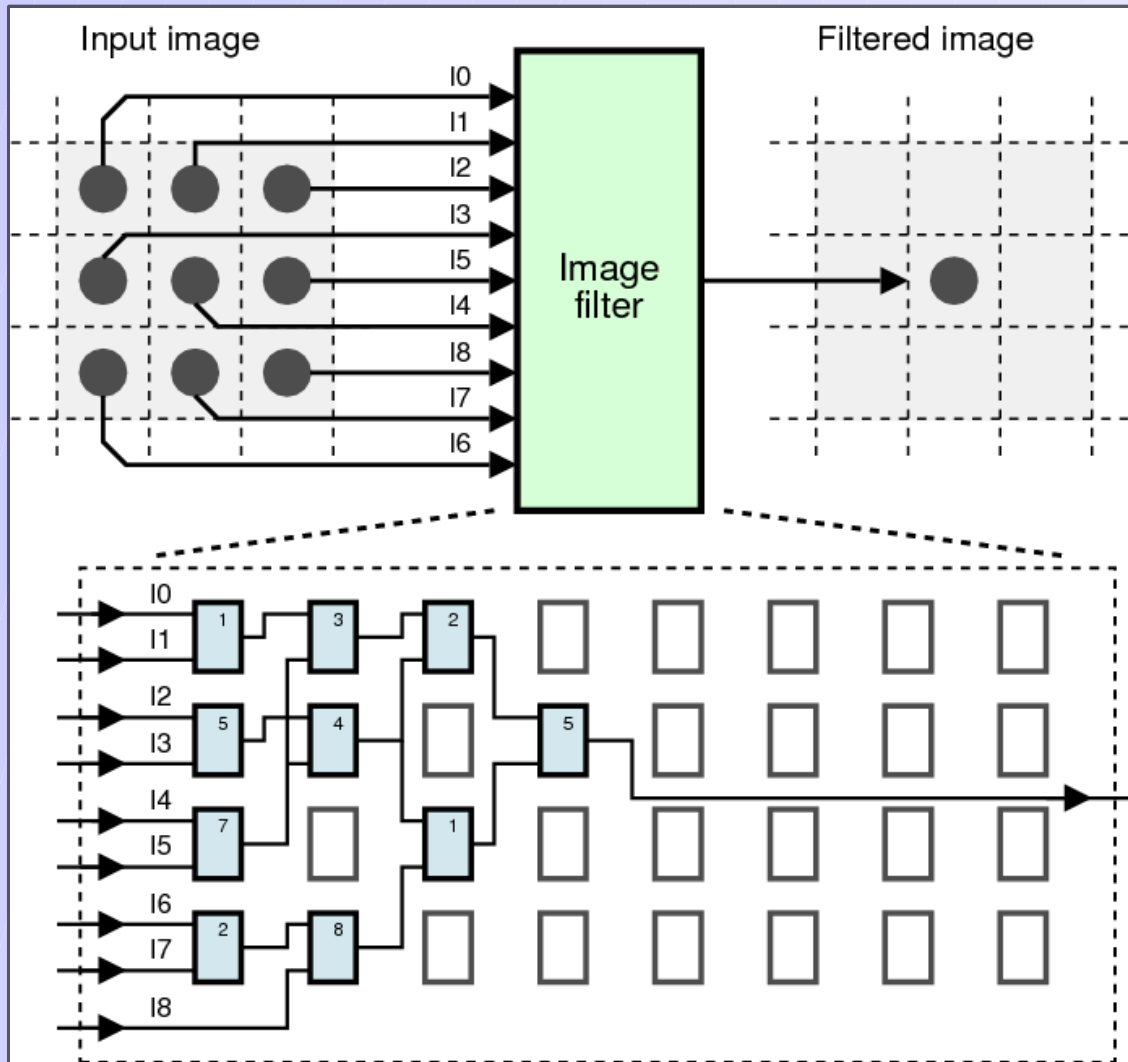
    // copy device memory to host
    CUDA_SAFE_CALL( cudaMemcpy( arr3i, d_odata, mem_size, cudaMemcpyDeviceToHost) );

    // check for errors, deallocate memory, ...
}
```



- This paradigm was introduced by J. Miller and P. Thomson in 1999.
- Represents a candidate program / circuit by an rectangular matrix of programmable elements.
- Function of an element and its links with other elements can be changed – this is the task of a genetic algorithm to find a desired configuration.
- Can represent any circuit at a desired level gate/function, transistor.

# CGP for Image Filter Evolution



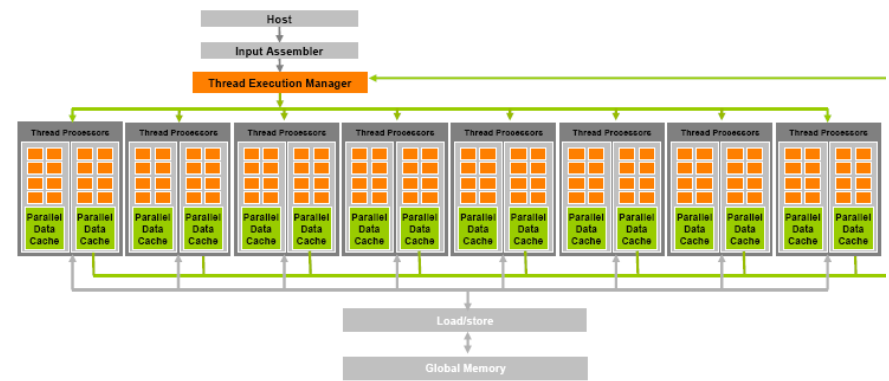
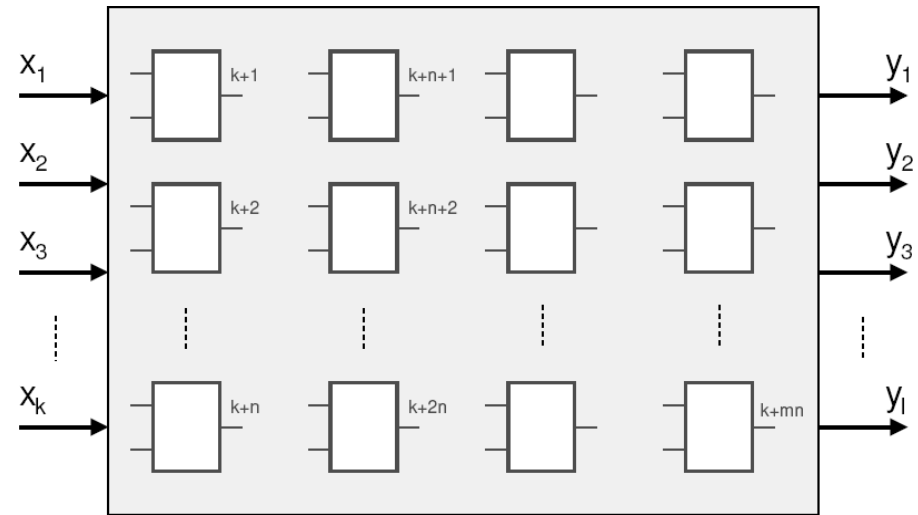
- Input image is scanned with an square mask in which the elements are connected to filter inputs.
- The filter output is then written to an appropriate place in the output image.
- The whole input image is scanned, filtered and results are written to the output image.



- Each filter can be separately processed by a single thread processor in each block. (ineffective, easy to implement)
- 
- All thread processors in each block can evaluate different filters. (an interpreter running on each thread processor is required – leads to plenty of NOPs, large populations have to be evaluated in order to use all resources)

## ● Pipelined evaluation

- one multiprocessor evaluates one column, the results pass to the next multiprocessor
- synchronization between multiprocessors is very difficult
- limited interconnection parameter (l-back = 1)
- use of global memory for communication
- real implementation would be very slow



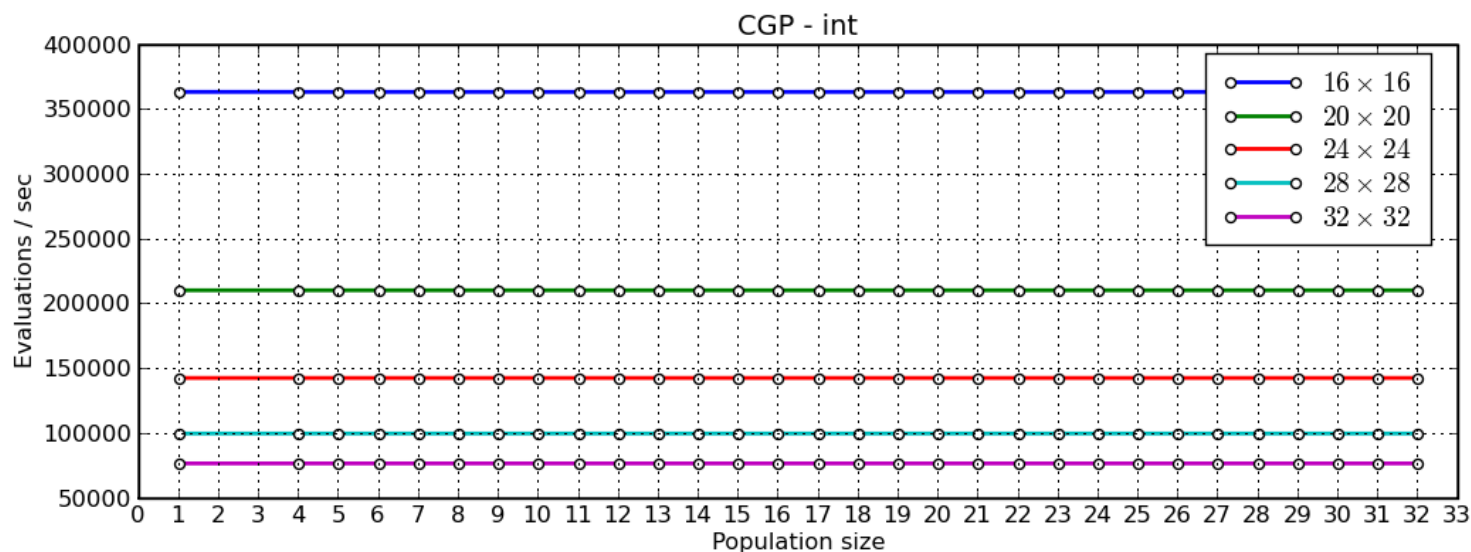
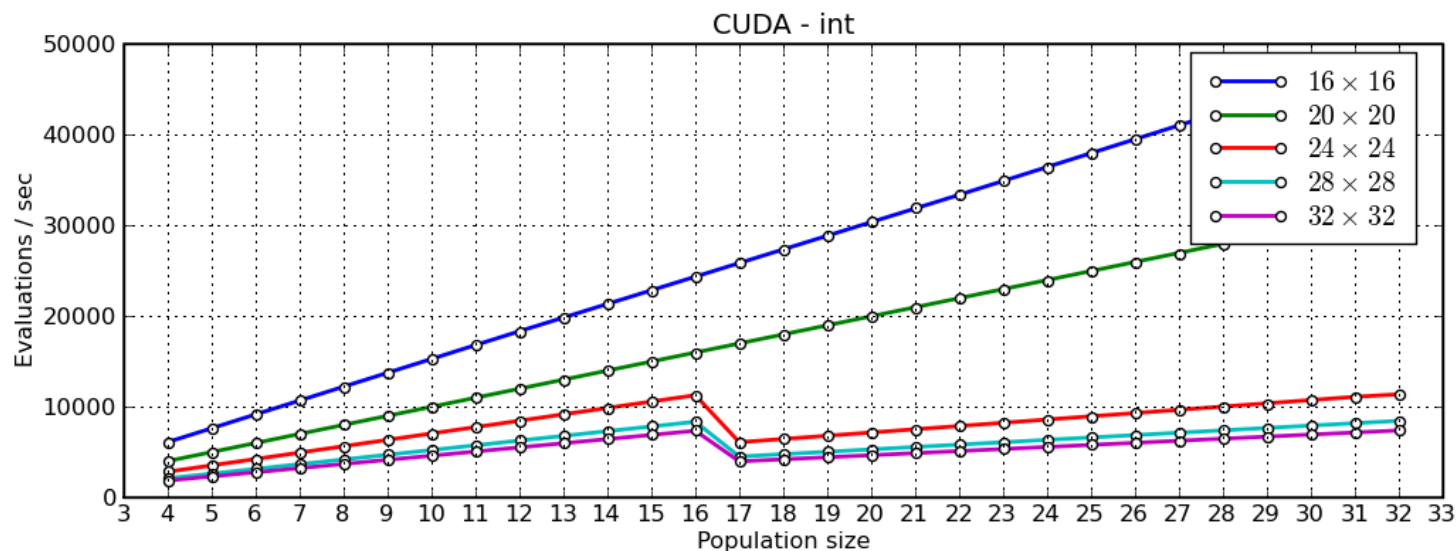
- **column-wise evaluation**
  - each thread processor evaluates a single chromosome column in a single step
  - fast shared memory is used for thread communication
  - all columns are evaluated in a loop cycle on the same multiprocessor
  - whole population can be evaluated in parallel
  - this implementation is used for experiments

- **experimental settings**

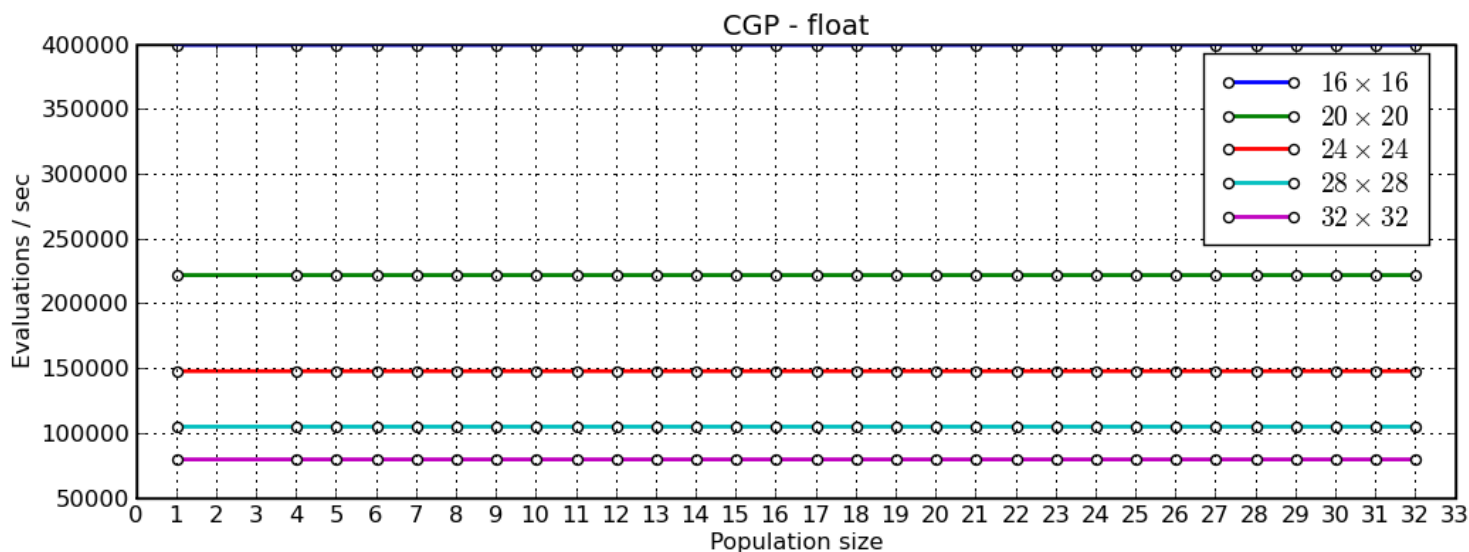
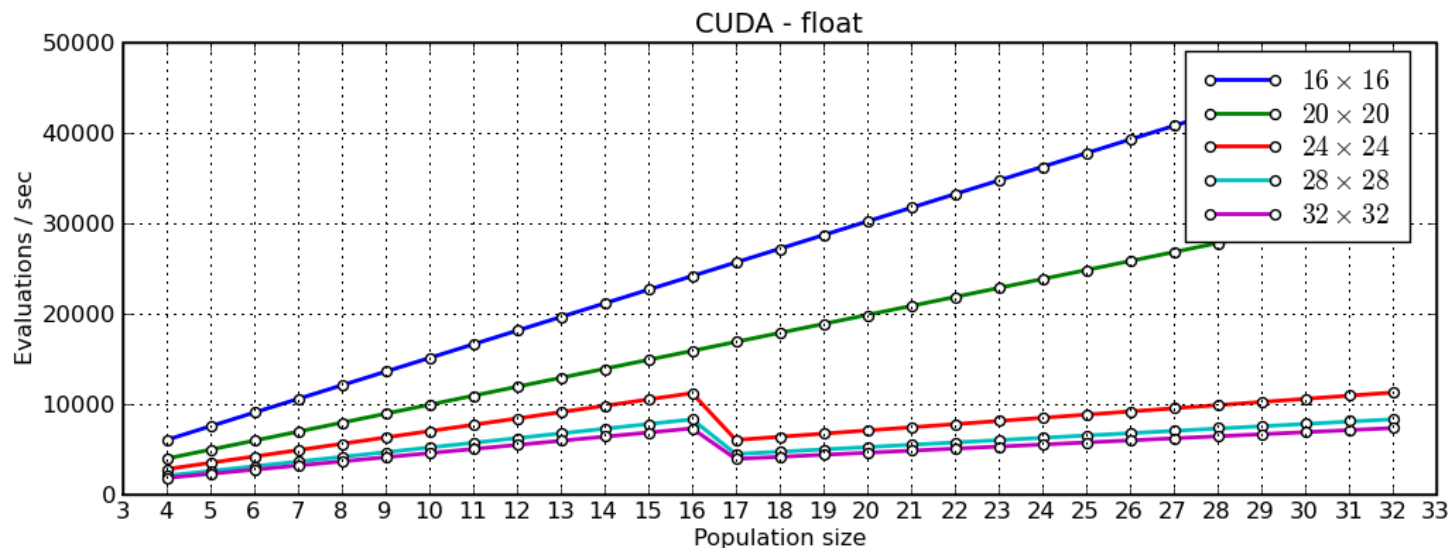
one whole population was evaluated  
1000 times on GPU  
4000000 times on CPU  
average data were recorded

population size = number of grids  
number of chromosome rows = number threads

# Integer CGP evaluation



# Float CGP Evaluation



```
__global__ void kernel(int * idata, int * odata) {
    allocate_shared_for_input_chromosomes();
    allocate_shared_variables();
    copy_chromosomes_to_local_shared(idata);
    prepare_chromosome_inputs();

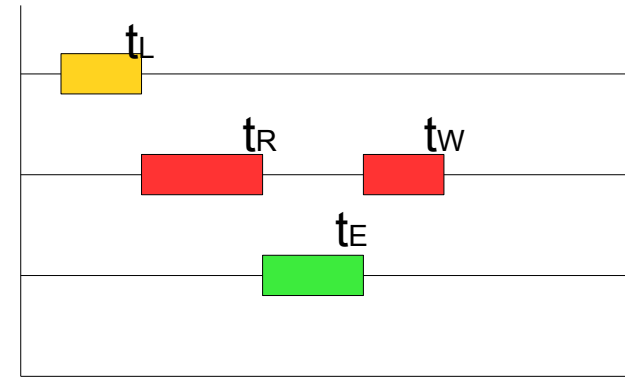
    for (c = 0; c < CHROMOSOME_COLUMNS; c++) {
        register unsigned int c_row = c * CHROMOSOME_ROWS;
        register unsigned int thread_index = (threadIdx.x + c_row) *
                                             (NODE_INPUTS + 1);

        outcomes[threadIdx.x + c_row + CHROMOSOME_INPUTS] =
            nodeFuncD(chromosome[thread_index],
                    outcomes[chromosome[thread_index + 1]],
                    outcomes[chromosome[thread_index + 2]]);

        __syncthreads();
    }

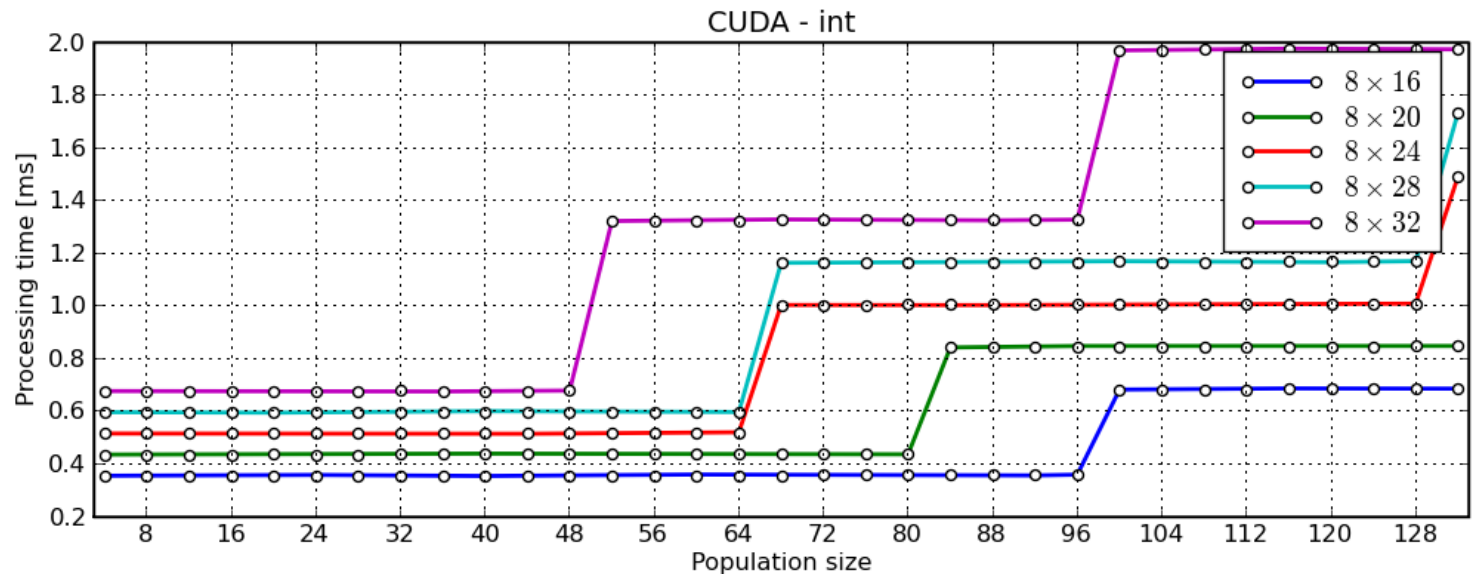
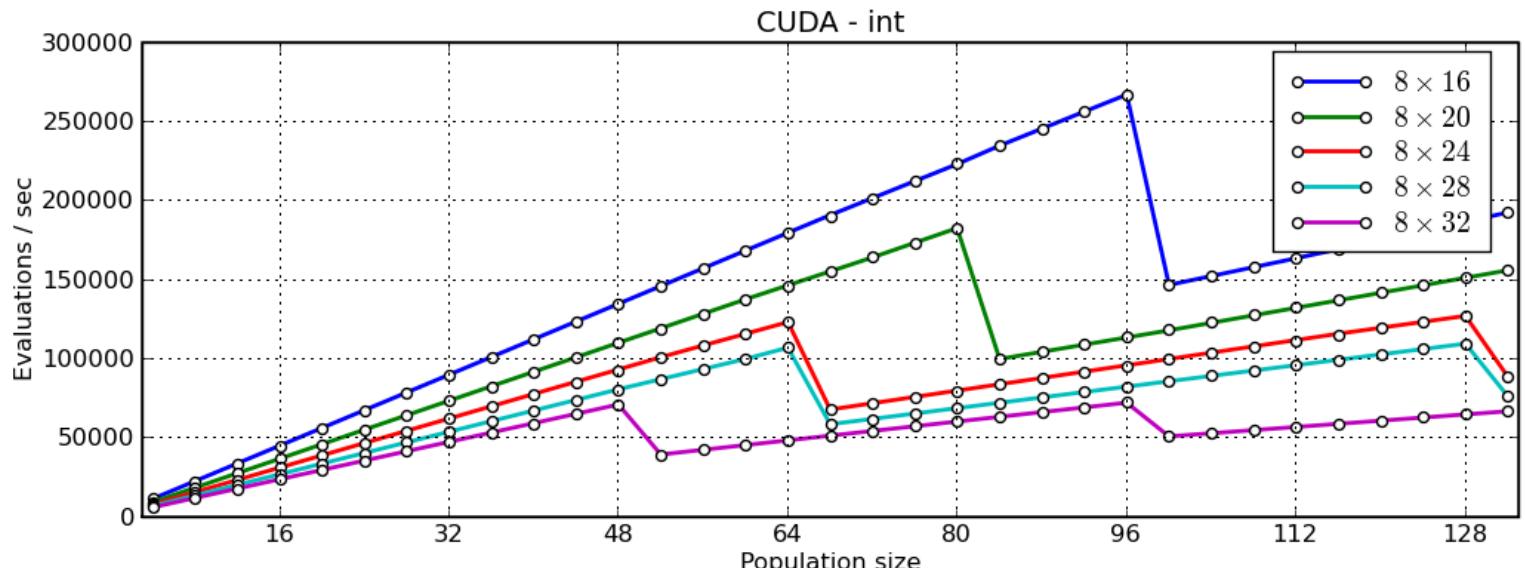
    copy_results_from_local_shared(odata);
}
```

- **kernel loading time**
- depends on chromosome length (loop enroll)
- **read/write global**
- depends on population and chromosome size
- **chromosome evaluation**
- good scalability
  
- **peak performance**  
 $(t_L + t_R + t_W) / t_E \rightarrow 0$

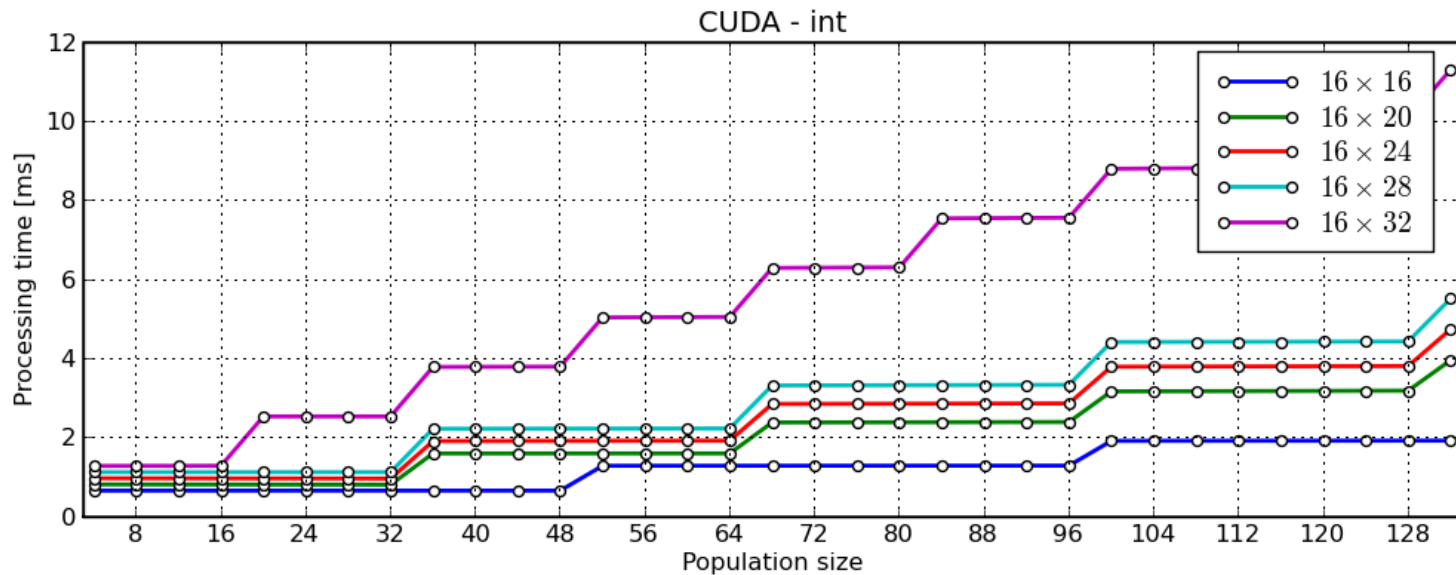
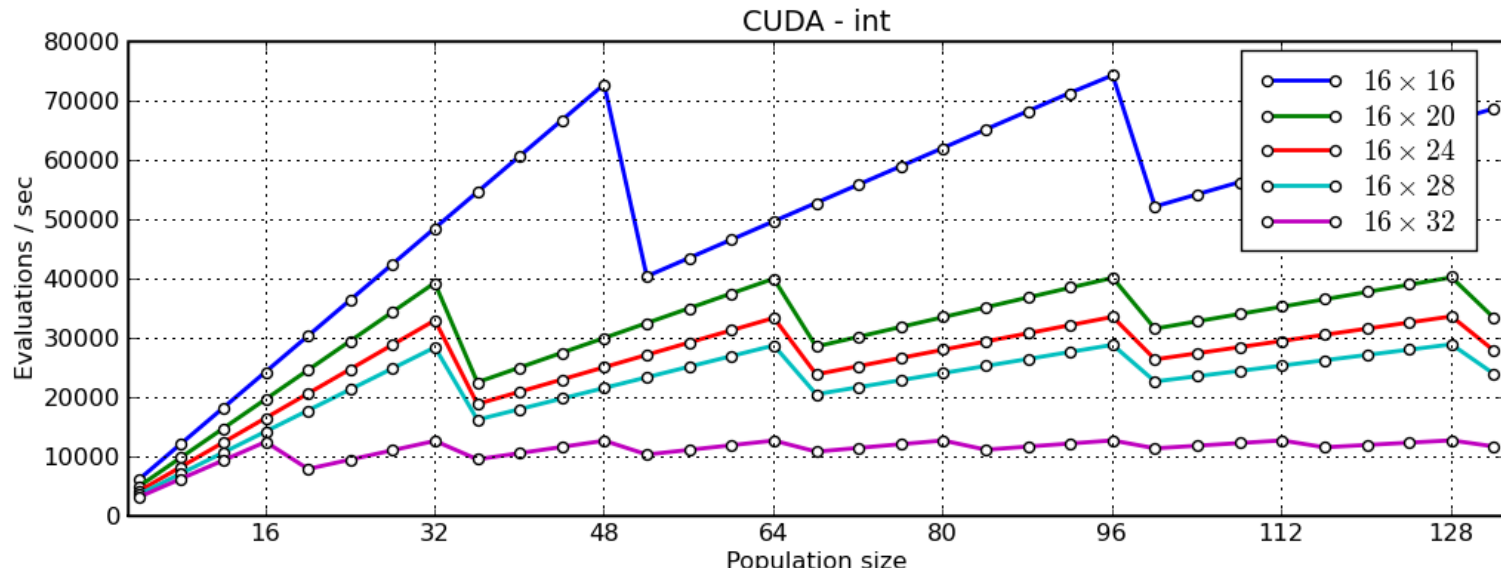


$$t_L + t_E + t_R + t_W = t_K$$

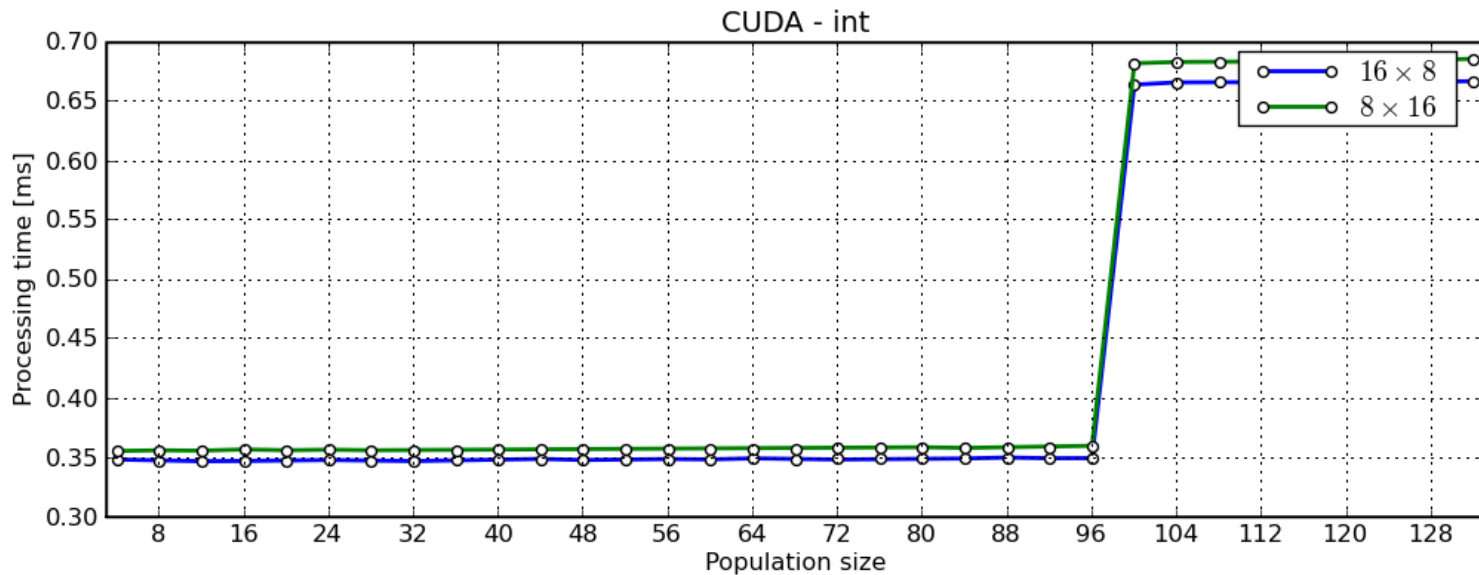
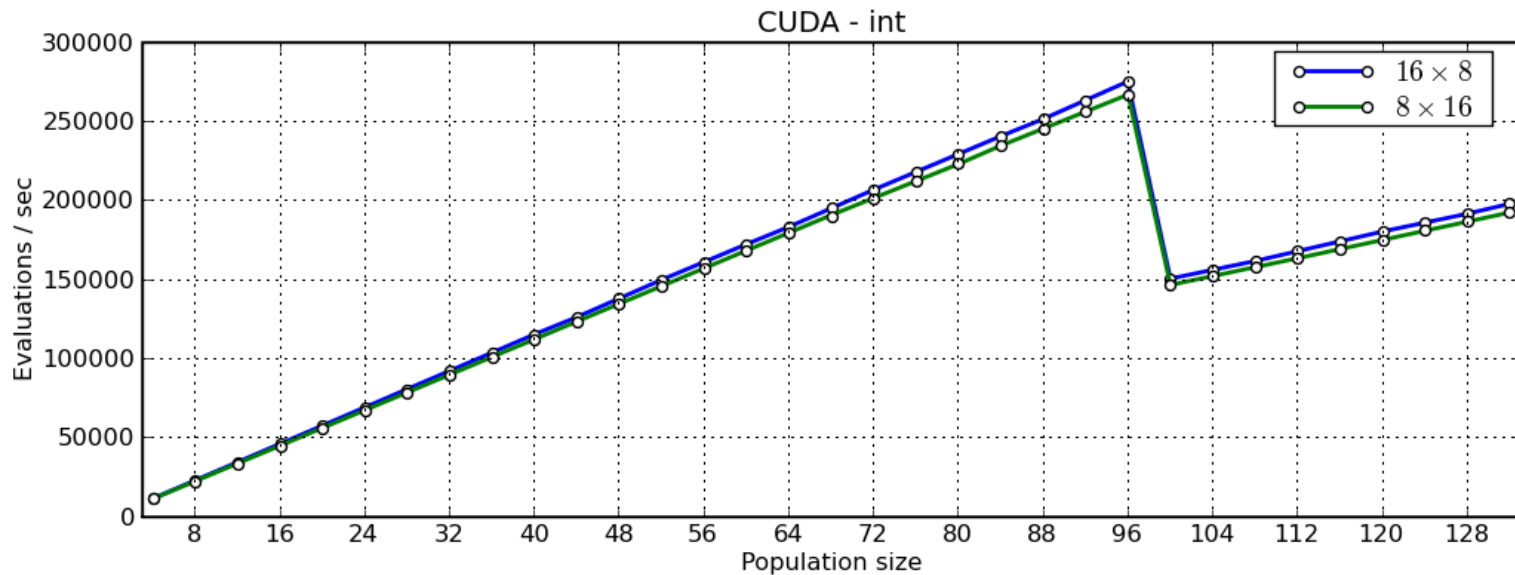
# Integer Performance (8xX)



# Integer Performance (16xX)



# Integer Performance (8x16 vs 16x8)



- the use of the global device memory is painfully slow in comparison with the shared memory
- the use of the texture memory is also not of a great benefit
- the small shared memory is a limitation for evaluated genome/population size
- the device is accessed via cuda library calls; the library can perform some optimizations (strange code behaviour)



Thank you for attention!