A Support Vector Machine Implementation on NVIDIA Graphic Processing Units

Daniel Llamocca
Ingo Steinwart

Outline

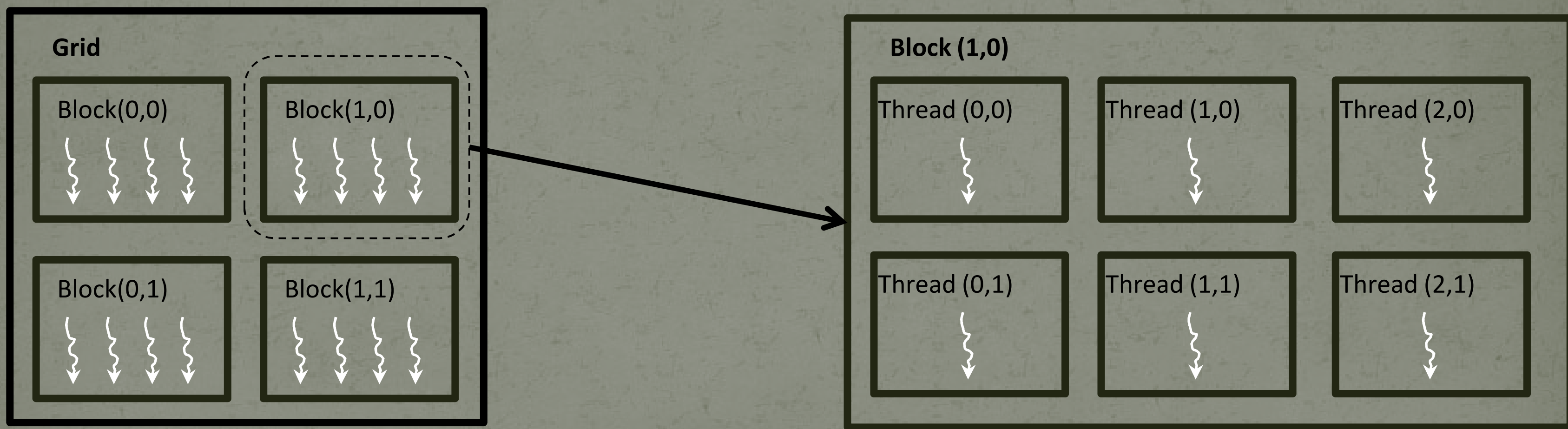
- Objectives
- Basics of CUDA (Compute Unified Device Architecture) Programming Model
- Parallelization of the algorithms
- Issues and performance guidelines
- Results
- Conclusions

Objectives

- Expose and exploit parallelism of the following routines:
  - Support Vector Machine testing (classification)
  - Support Vector Machine training
  - Validation phase of SVM training
- GPU implementation of the parallel algorithms.
- Obtain significant speedup improvements.

Basics of CUDA Programming Model

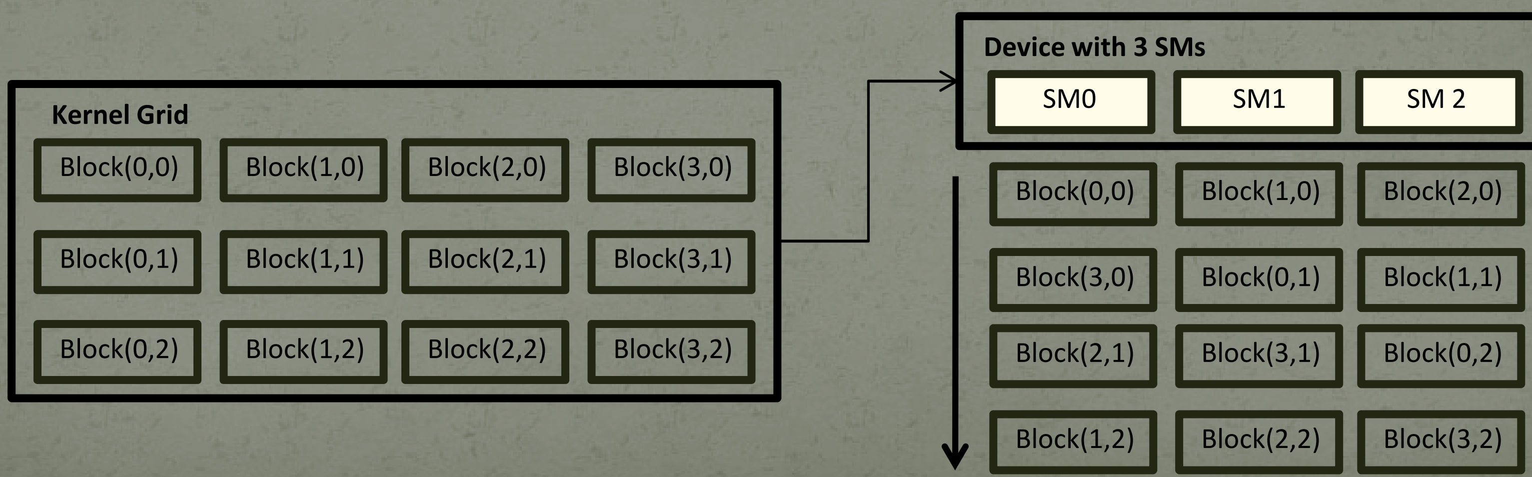
- CUDA: Computing architecture that leverages the parallel computing engine on NVIDIA GPUs. It allows developers to use C language.
- Parallelism is described by a grid that consist of blocks, each block having a number of threads:



Basics of CUDA Programming Model

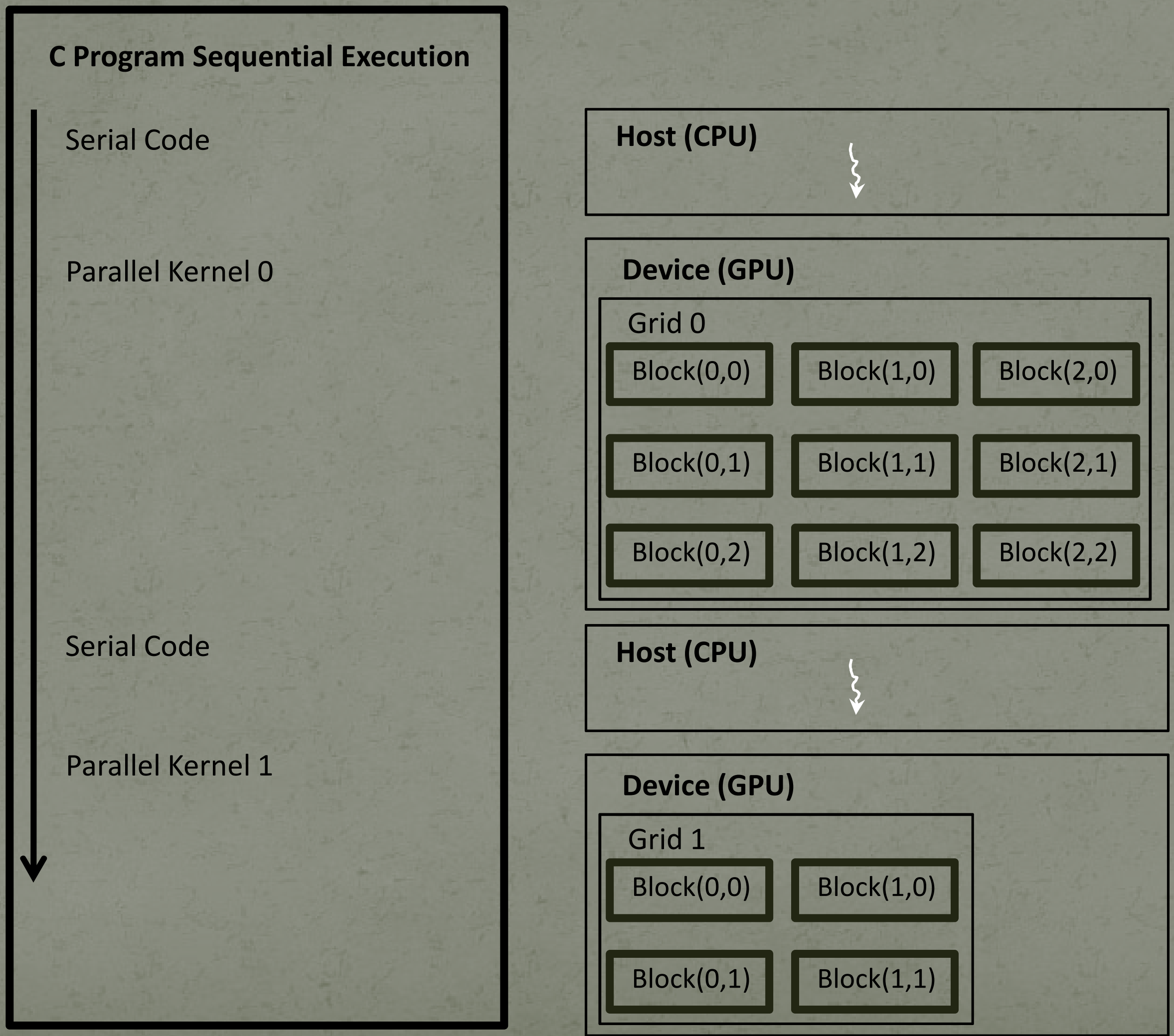
- Grid Size and Block Size limitation:

  NVIDIA GTX 295: 512 threads per block (1D, 2D, or 3D block). 65535 blocks per grid (1D or 2D grid)
- All threads in a block run in parallel. A limited number of blocks can be run in parallel. NVIDIA GTX 295: 30 SM (streaming multiprocessors) ➔ up to 30 blocks can execute concurrently, i.e. 30x512 = 15360 threads.



Basics of CUDA Programming Model

- Kernel: Code executed on the GPU:



Basics of CUDA Programming Model

- GPU Memory: There are 4 basic kind of memories:
  - Global Memory: Non-cached and high latency memory. It is available to all threads and is able to communicate with CPU memory.
  - Shared Memory: Faster than global memory. GTX295: 16384 bytes of shared-memory per thread block.
  - Constant Memory, Texture Memory.
- Streams: Allow CPU to GPU communication to overlap with GPU kernel execution.

Parallelization of the algorithms SVM testing

- The following function is to be computed:

$$f(x_i) = \sum_{k \in SV} y_k \alpha_k K(xt_k, x_i), \forall i \in test\ dataset$$

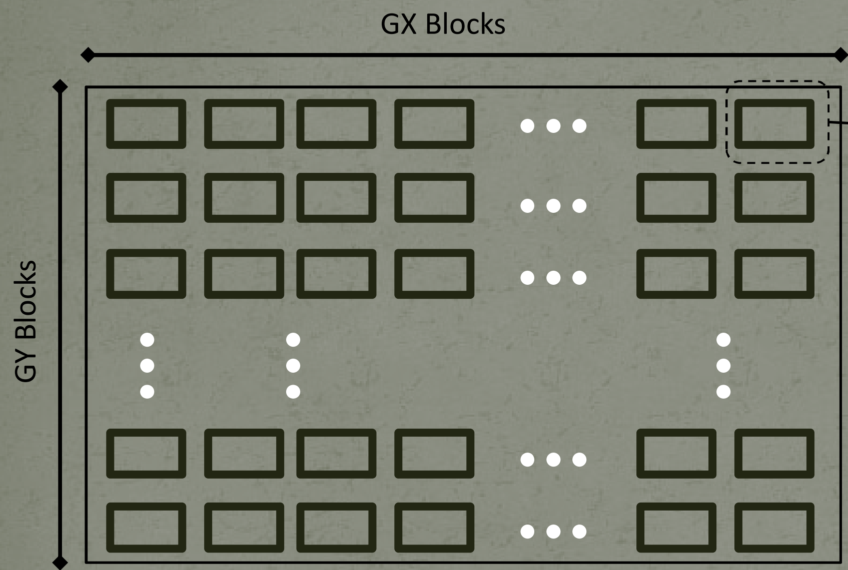
$$x_i : test\ dataset\ sample, i = 1 : N$$

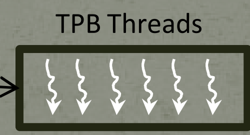
$$xt_k : train\ dataset\ sample, y_k \alpha_k : train\ coefficients, k = 1 : SV$$

- Parallelization: 2 Kernels:
- Compute individual summations: Each thread will compute SPT sums of products.
- Compute final function value: Each thread will add up the previous values to get the final function value.

Parallelization of the algorithms SVM testing

- First Kernel: Each thread computes SPT sums of products. (SPT = 4, 8, 12, …) $\sum_{k=0}^{SPT-1} y_k \alpha_k K(xt_k, x_i)$





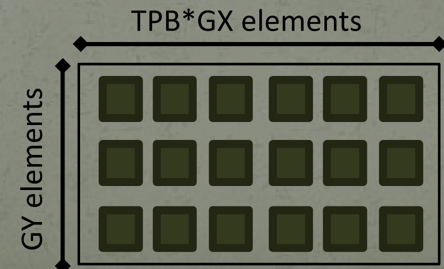
$TPB: threads\ \ per\ block(256, 512)$

$GY: test\ \ samples\ \ processed\ by\ a\ grid$

$GX * GY \leq 65535$

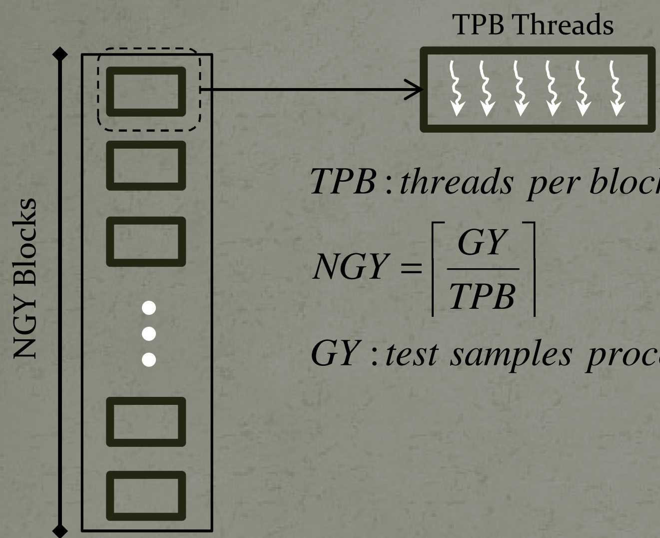
$$GX = \left\lceil \frac{SV}{TPB * SPT} \right\rceil, GY = \left\lfloor \frac{65535}{GX} \right\rfloor$$

- 1st kernel output: A vector of SV/SPT elements for every one of the GY samples. Vectors are stored in global memory



Parallelization of the algorithms
SVM testing

- Second Kernel: Each thread will add up the previous values to get the final function value for a test sample.



$TPB: threads\ per\ block (256, 512)$

$$NGY = \left\lceil \frac{GY}{TPB} \right\rceil$$

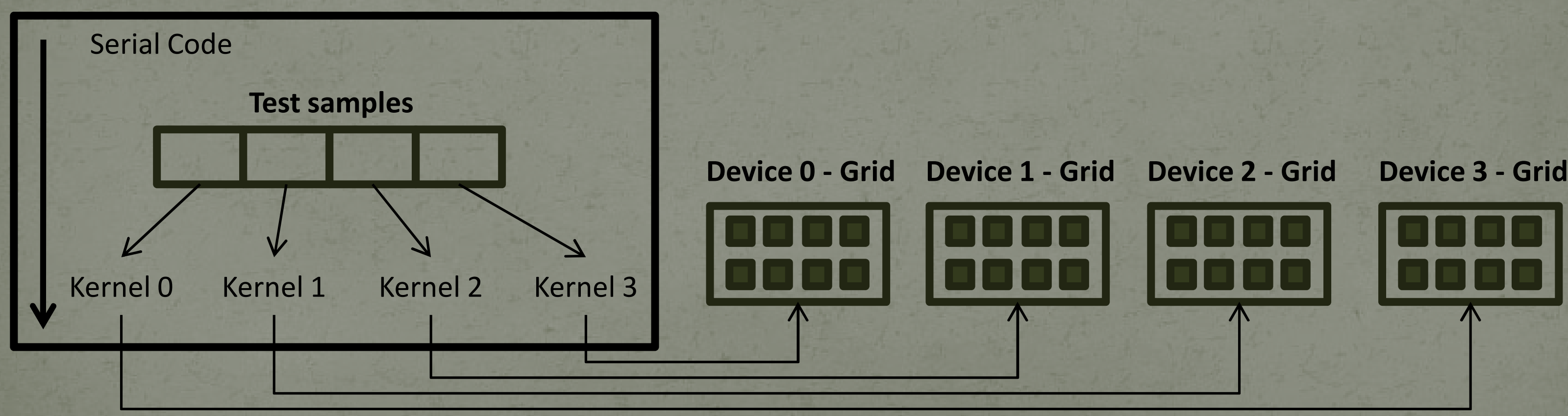
$GY: test\ samples\ processed\ by\ a\ grid$

- 2^{nd} kernel output: A vector of GY function values

- Since N test samples might be way larger than GY, the 2 kernels have to be run NC times. $NC = \left\lceil \frac{N}{GY} \right\rceil$

Parallelization of the algorithms
SVM testing

- 'frings.lanl.gov' machine: It contains 2 NVIDIA GTX295 boards, each one with 2 GPUs.
- Therefore, we further parallelized the algorithm by running 4 CPU threads, each CPU thread run a GPU device code independently. The number of test samples is divided in 4 chunks, each chunk is processed by a GPU device.



Issues and Performance Guidelines

- GY, GX, NGY, NC involve floor and ceil operations. Take NC for instance: at the last run, the number of samples processed has to be smaller than GY, otherwise there will be wasted computation. This introduces more flow control that impacts the overall performance.
- Call to CPU thread that enables GPU operation: There is an overhead of 80 ms. Try to keep those calls as low as possible.
- Global GPU memory is not cached, thus this is the slowest memory. In our case, shared memory was deemed impractical and global memory was used.
- Communication between CPU memory and GPU global memory is the slowest process. Do it when really necessary, and transfer one big chunk instead of several small chunks.

Results

- C10-Oct14 Dataset: 11 decision functions

		Test size					
		815,000			**50,000**		
Support Vectors	**1000**	4 CPU cores 274.6 s	1 GPU:	36.8 s	4 CPU cores 19.2 s	1 GPU:	3.7 s
			2 GPUs:	20.3 s		2 GPUs:	3.5 s
			3 GPUs:	15.3 s		3 GPUs:	4.6 s
			4 GPUs:	13.2 s		4 GPUs:	6.2 s
	2000	4 CPU cores 462.5 s	1 GPU:	78.1 s	4 CPU cores 29.6 s	1 GPU:	6.2 s
			2 GPUs:	41.0 s		2 GPUs:	4.8 s
			3 GPUs:	29.1 s		3 GPUs:	4.9 s
			4 GPUs:	23.6 s		4 GPUs:	6.2 s

Conclusions

- We achieve speed-up of about 20x is large datasets. In small datasets, the speed-up is about 6x. This suggests that the larger the datasets, the better the speed-up improvement.
- Consider that we are using double floating point arithmetic (64 bits). This requires twice the amount of registers in the GPU than in the case of single floating point arithmetic; if the registers do not fit in the register space, the compiler will place the 'registers' in the slow global memory.