Workshop: School of Data Science and Computation Thinking, Stellenbosch University

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THURST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)

LECTURE 5 (4 March 2025)

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

Zoom link:

`https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1`

Course webpage:

`http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation`
`https://github.com/martinabucher/cudaCourse`

Lecture 5 Outline

1. Probing Device Properties (continued - complete slides)
2. Reduction
3. Challenges of Sharing Data: Mutual Exclusion
4. Livelocks, Deadlocks and Race Conditions
5. Atomic Operations
6. Tuning Matmul
7. Memory Bound and Computation Bound Problems
8. Exercise: Parallel Sorting

PROBING DEVICE PROPERTIES

From command line and using host code.

```
bucher@stoertebeker:~$ nvidia-smi
Tue Mar  4 08:58:15 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 4090        Off | 00000000:01:00.0 Off |                  Off |
|  0%   41C    P8              19W / 450W |     11MiB / 24564MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1442      G   /usr/lib/xorg/Xorg                            4MiB |
+---------------------------------------------------------------------------------------+
bucher@stoertebeker:~$ nvidia-smi -q

==============NVSMI LOG==============

Timestamp                                 : Tue Mar  4 08:58:23 2025
Driver Version                            : 535.183.01
CUDA Version                              : 12.2

Attached GPUs                             : 1
GPU 00000000:01:00.0
    Product Name                          : NVIDIA GeForce RTX 4090
    Product Brand                         : GeForce
    Product Architecture                  : Ada Lovelace
```

https://github.com/martinabucher/cudaCourse/blob/main/printDeviceProperties.cu

```
#include <stdlib.h>
#include <stdio.h>
#include "helper.cuh"

int main(){
  cudaDeviceProp prop;
  int count;
  errCheck(cudaGetDeviceCount (&count));
  if(count==0){printf("Error: No CUDA enabled devices found.\n"); exit(-1);}
  int device=0;
  errCheck(cudaGetDeviceProperties (&prop, device));
  printf("clockRate                  = %d (%g)\n", prop.clockRate
                               (double) prop.clockRate                 );
  printf("l2CacheSize                = %d (%g)\n", prop.l2CacheSize
                             , (double) prop.l2CacheSize               );

                                   ***

  return 0;
}
// https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html
// https://docs.nvidia.com/cuda/cuda-runtime-api/
//          structcudaDeviceProp.html#structcudaDeviceProp
```

```
clockRate                   = 2550000 (2.55e+06)
l2CacheSize                 = 75497472 (7.54975e+07)
major                       = 8 (8)
minor                       = 9 (9)
maxBlocksPerMultiProcessor  = 24 (24)
maxGridSize[0]              = 2147483647 (2.14748e+09)
maxGridSize[1]              = 65535 (65535)
maxGridSize[2]              = 65535 (65535)
maxThreadsDim[0]            = 1024 (1024)
maxThreadsDim[1]            = 1024 (1024)
maxThreadsDim[2]            = 64 (64)
maxThreadsPerBlock          = 1024 (1024)
maxThreadsPerMultiProcessor = 1536 (1536)
memPitch                    = 2147483647 (2.14748e+09)
memoryBusWidth              = 384 (384)
memoryClockRate             = 10501000 (1.0501e+07)
multiProcessorCount         = 128 (128)
name[256]                   = NVIDIA GeForce RTX 4090
persistingL2CacheMaxSize    = 51904512 (5.19045e+07)
regsPerBlock                = 65536 (65536)
regsPerMultiprocessor       = 65536 (65536)
sharedMemPerBlock           = 49152 (4.915200e+04)
sharedMemPerBlockOptin      = 101376 (1.013760e+05)
sharedMemPerMultiprocessor  = 102400 (1.024000e+05)
totalConstMem               = 65536 (6.553600e+04)
totalGlobalMem              = 25393692672 (2.539369e+10)
warpSize                    = 32
```

What is the class (difficulty) of the following problem on a parallel processor?

- Problem
$$\sum_{n=0}^{2^D} f(n)$$
where $N, D \to \infty$.
- How would we implement it on an idealized PRAM? On CUDA? What kind of special parallel support is needed?

Reduction (I)

"<u>Reduction</u>" = Parallel Computation of Sums

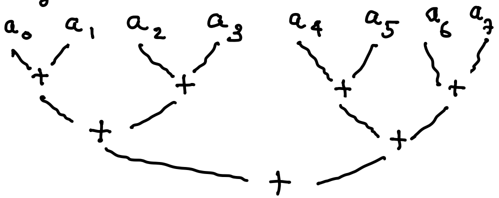
How long does it take to compute the following?

$$a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$$

Associativity allows to regroup

$$\{(a_0 + a_1) + (a_2 + a_3)\} + \{(a_4 + a_5) + (a_6 + a_7)\}$$

Diagramtically:



$$(\text{Time}) \sim \lceil \log_2 N \rceil$$

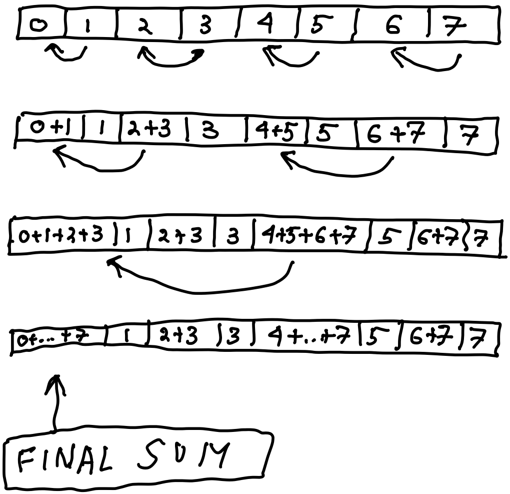
Reduction (II)

Reduction applies to any binary associative operator. E.g.

$+$ $\times$

AND OR

Reduction (II)



Reduction Kernel (summation)

```
#include <stdio.h>
__global__ void testReduceSum(float* s, const int n, float *result);

__device__ void reduce_sum(float *s, const int n, float *sum){
  if (n == 1){   // eliminate trivial case
     *sum = s[0];
  }
  int index = threadIdx.x + blockDim.x * blockIdx.x; int step=1;
  while(1){
    if (2*step >= n ) break;
    step*=2;
  }
  if (index == 0 )
     printf("step %d\n",step);
  int n_max=n;
  while(1){
    if ( index + step < n_max )
      s[index]+=s[index+step];
    __syncthreads();
    n_max=step;
    if ( step==1 ) break;
    step/=2;
  }
  *sum=s[0];
}
```

```
int main(void){
  const int thread_dim=10;
  const int block_dim =1;
  int n=100;
  size_t size=n*sizeof(float);
  float* s_d;
  float *result_d;
  float result_h;
  cudaMalloc((void **) &s_d, size);
  cudaMalloc((void **) &result_d,sizeof(float));
  float s_h[size];
  float local_sum=0.;
  for(int j=0;j<n;j++){
    s_h[j]=(float) (j+1);
    local_sum+=s_h[j];}
  printf("Expected sum is equal to %f \n", local_sum);
  cudaMemcpy(s_d, s_h, size, cudaMemcpyHostToDevice);
  testReduceSum<<<block_dim,thread_dim>>>(s_d,n,result_d);
  cudaMemcpy(&result_h, result_d, sizeof(float), cudaMemcpyDeviceToHost);
  printf("CUDA sum is equal to %f \n", result_h);
}
```

```
bash-4.2$ ./my_reduce
Expected sum is equal to 5050.000000
step 64
CUDA sum is equal to 5050.000000
bash-4.2$
```

Challenges of Sharing Data

- Suppose that we do a lot of work by different cores and the end result is to be sum of all the partial results. Suppose moreover that the most of the work is in the computation of the partial results and not in the computation of the sum. This differs from our discussion of reduction where the object was to attain maximum speed for computing a sum.
- We want to carry out an operation of the sort

  ```
  sum=sum+new_contribution;
  ```

  where the variable sum is shared between many processes.
- One of the problems is that the above operation when translated into machine instructions involves more than one step, even though in C/C++ it appears as a single statement.
  1. Retrieve sum (from main memory) to a register.
  2. Add new_contribution to the value of sum in the register.
  3. Write the new value back to memory.
- This needs to occur in such a way that only one processor at a time accesses the variable sum.
- One solution is to use a lock aka mutex=(mutual exclusion).

```
#include <chrono>
#include <iostream>
#include <map>
#include <mutex>
#include <string>
#include <thread>

std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;
void save_page(const std::string& url){
    // simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    std::lock_guard<std::mutex> guard(g_pages_mutex);
    g_pages[url] = result;}

int main(){
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join();
    t2.join();
    // safe to access g_pages without lock now, as the threads are joined
    for (const auto& [url, page] : g_pages)
        std::cout << url << " => " << page << '\n';}
```

Output:

```
http://bar => fake content
```

C++ interlude

- Here the C++ main program creates two threads running on different cores.
- In C++ the mantra is "RAII" or "Resource Acquisition Is Initialization". The idea is that the root of all evil is uninitialized variables or forgetting to release ressources once they are not needed.
- Thus when a variable is declared a "constructor" is called that initializes the variable, and when the variable falls out of scope, a destructor" is called to carry out any needed cleanup action.
- We already recounted how the founders of Java considered pointers the "root of all evil", replacing pointers with references and garbage collection. RAII is in part Bjarne's Stroustrup's reply to this criticism.
- It is worth looking at

  `https://en.cppreference.com/w/cpp/language/raii`

  where this ideology has been made part of the C++ standard library documentation.

The bad and the good according to the C++ gurus

```
std::mutex m;
void bad(){  // THIS IS HOW ONE WOULD PROGRAM THIS IN OLD-STYLE C
    m.lock();  // acquire the mutex
    f();       // if f() throws an exception, the mutex is never released
    if (!everything_ok())
        return; // early return, the mutex is never released
    m.unlock(); // if bad() reaches this statement, the mutex is released
}

void good(){ // THIS IS THE C++ WAY OF DOING THE SAME
    std::lock_guard<std::mutex> lk(m);
        // RAII class: mutex acquisition is initialization
    f();       // if f() throws an exception, the mutex is released
    if (!everything_ok())
        return;// early return, the mutex is released
}        // if good() returns normally, the mutex is released
```

One cannot forget to include releasing the lock when writing the program, because this is done implicity at the end of subroutine.

https://en.cppreference.com/w/cpp/language/raii

Not everyone loves C++

MOMMY, WHO MAKES CHRISTMAS TOYS?
THE ELVES!
WOW REALLY? EVEN VIDEO GAMES?
SURE!
OMG WTF IS THIS SHIT.
C++ FOR DUMMIES
JEGO

Source: Reddit

What can go wrong (I): Deadlocks (part 1)

- In general, function calls to acquire locks are **blocking**, which means that they do not return until the lock is acquired. In other words, if someone else has the lock, the routine waits. Normally, after a while the thread would be swapped out of the CPU or core, assuming that there are more threads than cores.
- Another option is for the function call attempting to acquire the lock to return with a failure status.
- Worst Case Scenario (but not so uncommon if one is not careful)
  Consider the two processes running on different threads:

```
process_1(){
   get_lock_A();
   get_lock_B();
   do_work_1();
   release_lock_B();
   release_lock_A();
}

process_2(){
   get_lock_B();
   get_lock_A();
   do_work_1();
   release_lock_A();
   release_lock_B();
}
```

What can go wrong (I): Deadlocks (part 2)

- The two (asynchronous) streams may interleave arbitrarily. For example, the following chronological sequence is possible:

  ```
  process_1 acquires lock A.
  process_2 acquires lock B.
  ```

- Now `process_1` is waiting for lock B to be released and `process_2` is waiting for lock A to be released. Both get_lock calls are blocked, each waiting for something to happen that will never happen.
- The correctness of a parallel program relies on it being able to run correctly and finish for all possible interleaving of the execution chronology.
- What are some solutions to this problem?

Some more considerations:

- "Interleaving" not mean what you think it does.
- The simplest interpretation would be to order sequences of code (say for two processes A and B: as A1, A2,....,AM and B1, B2, ..., BN and enumerate of sequences of length M+N where all the As appear exactly once respecting the A order and all the Bs appear exactly once respecting the B order.
- But these operations may not be atomic, so finer-grained overlap or simultaneity may occur.
- An example is

  ```
  sum=sum+term;
  ```

Livelock (I)

```
process_1(){
  bool isHungry=true;
  while(isHungry){
    if (get_lock_A(){
       if (get_lockB() ){
         do_work_1();
         is_hungry=false;
         release_lock_B()
       }
    } else {
      release_lock_A();
    }
  }
}

process_2 is the same except that
B and A are interchanged and the work is
different.
```

Livelock (II)

An infinite loop is created, and nothing gets done. Each tries to defer to the other, but the end result is not successful.

```
process 1 acquires lock A
process 2 acquires lock B
process 1 fails to get lock B
process 2 fails to get lock A
process 2 releases lock B
process 1 releases lock A

process 1 acquires lock A
process 2 acquires lock B
process 1 fails to get lock B
process 2 fails to get lock A
process 2 releases lock B
process 1 releases lock A

ad infinitum
```

Race Conditions

- The final result depends on how the asynchronous execution is interleaved.
- The outcome of the race is not necessary predictable. It may depend on the hardware or on the data.
- These kinds of errors are hard to find and may pop up after the software appears to have been succesfully tested.

Atomic Operations (I): Atomic CAS (Compare and Swap)

```
int compare_and_swap(int *destination_address, int expected_value,
int desired_value);

The following operation is carried out atomically.

If the value at the destination address value is equal to the expected value, i
by the desired value, which is also value returned. Otherwise, the
present value is returned. [swap may succeed or fail].

Usage exam:

int sum=0;
int expected_value;
do { expected_value=sum;
     desired_value=expected_value+partial_sum;
} while(desired_value!=compare_and_swap(&sum, expected_value, desired_value);

When there is contention at least one thread succeeds.
```

Atomic Operations in CUDA (I)

See Chapter 7.14 of CUDA C++ Programming Guide for Documentation

```
https://docs.nvidia.com/cuda/cuda-c-programming-guide/
   index.html#atomic-functions
```

Atomic functions may have different scopes (system, device, block) entailing different overheads. See

```
https://nvidia.github.io/cccl/libcudacxx/extended_api/memory_model.html
```

Example: Atomic Add

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 *address, __half2 val);
__half atomicAdd(__half *address, __half val);
__nv_bfloat162 atomicAdd(__nv_bfloat162 *address, __nv_bfloat162 val);
__nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);
float2 atomicAdd(float2* address, float2 val);
float4 atomicAdd(float4* address, float4 val);
```

Atomic Operations in CUDA (II)

Compare and Swap
7.14.1.8. atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
unsigned short int atomicCAS(unsigned short int *address,
                             unsigned short int compare,
                             unsigned short int val);
```

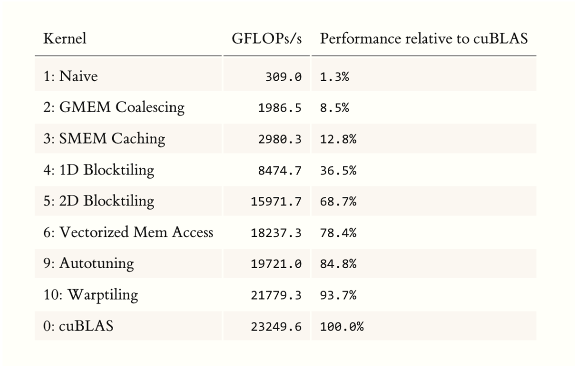
Atomic Operations in CUDA (II): Atomic CAS

Every atomic function may be expressed using the atomic CAS as an elementary building block. Here we implement an atomic add as an example.

```
__device__ void incrementFun(float *accumulator, const float increment){
  float old=*accumulator;
  while(true){
     float value=old+increment;
     int* value_int_ptr=(int*) &value;
     int value_int=*value_int_ptr;
     int* old_int_ptr=(int*) &old;
     int old_int=*old_int_ptr;
     int new_int=atomicCAS((int*) accumulator, old_int, value_int);
     if (new_int == old_int) break;
     float* new_float_ptr= (float*) &new_int;
     float newF=*new_float_ptr;
     old=newF;
  }
}
//https://github.com/martinabucher/cudaCourse/blob/main/atomicSumSquares.cu
```

An excellent blog on tuning matrix multiplication

https://siboehm.com/articles/22/CUDA-MMM



Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%

It is well worth studying this blog carefully.

Best Case Scenario

See Appendix A of the following document

https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

for the peak capabilities of the GeForce RTX 4090 GPU.

Some relevant numbers extracted from document:

Peformance Parameter	Value
GPU Boost Clock (MHz)	2520
Peak FP32 TFLOPS (non-Tensor)	82.6
Memory Bandwidth	1008 GB/sec
L1 Data Cache/Shared Memory	16384 KB
L2 Cache Size	73728 KB
Register File Size	32768 KB
Number of Cores	16384
Number of SMs	128

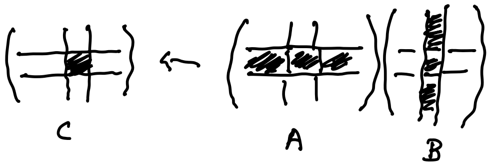
Some questions (exercises): Is matrix multiplication memory bound or computation bound? What speeds could be achieved under the most rosy scenario?

How does your program compare?

Block Matrix Multiplication (aka Tiling)

$$\left(\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & B_{12} & B_{13} \\ \hline B_{21} & B_{22} & B_{23} \\ \hline B_{31} & B_{32} & B_{33} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} C_{11} & C_{12} & C_{13} \\ \hline C_{21} & C_{22} & C_{23} \\ \hline C_{31} & C_{32} & C_{33} \end{array}\right)$$



Using shared memory

Variables declared with the keyword __shared__ are shared across a block.

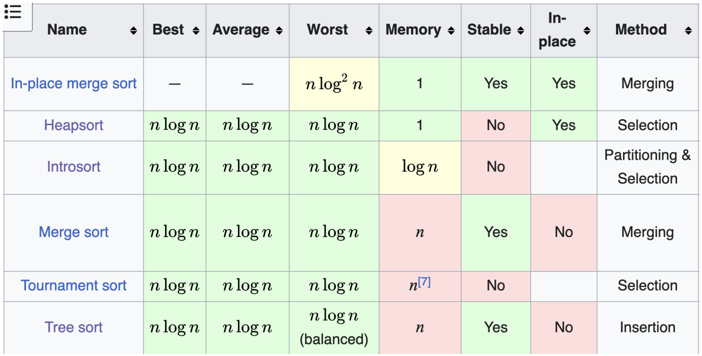
```
__global__ void matmul_tiled(int N, const float *A, const float *B, float *AB)


  __shared__ float tile_a[DIM_TILE*DIM_TILE], tile_b[DIM_TILE*DIM_TILE];

  int i=threadIdx.x, j=threadIdx.y;                               // indices with
  int i_block=DIM_TILE*blockIdx.x, j_block=DIM_TILE*blockIdx.y; // tile offsets
  float sum=0.;
  for(int n=0; n<DIM_GRID; n++){                                  // sum over blo
    int n_block=DIM_TILE*n;                                         // tile off
    // copy tiles
    tile_a[i*DIM_TILE+j]=A[DIM*(i+i_block)+(j+n_block)];
    tile_b[i*DIM_TILE+j]=B[DIM*(i+n_block)+(j+j_block)];
    __syncthreads();  // sync needed as each thread copies one element but elem
    for(int k=0;k<DIM_TILE;k++)   // accumulate a_tile@b_tile[i,j] to sum
      sum+=tile_a[i*DIM_TILE+k]*tile_b[k*DIM_TILE+j];
    __syncthreads();  // make sure all threads are done before proceeding to co
  }
  AB[(i+i_block)*DIM_TILE*DIM_GRID+(j+j_block)]=sum;
}
//https://github.com/martinabucher/cudaCourse/blob/main/matmulTiled_driver.cu
```

Exercise: GPU Sorting

```
https://en.wikipedia.org/wiki/Quicksort
https://en.wikipedia.org/wiki/Sorting_algorithm
```



Name	Best	Average	Worst	Memory	Stable	In-place	Method
In-place merge sort	—	—	$n\log^2 n$	1	Yes	Yes	Merging
Heapsort	$n\log n$	$n\log n$	$n\log n$	1	No	Yes	Selection
Introsort	$n\log n$	$n\log n$	$n\log n$	$\log n$	No		Partitioning & Selection
Merge sort	$n\log n$	$n\log n$	$n\log n$	n	Yes	No	Merging
Tournament sort	$n\log n$	$n\log n$	$n\log n$	n[7]	No		Selection
Tree sort	$n\log n$	$n\log n$	$n\log n$ (balanced)	n	Yes	No	Insertion

What can be achieved for an ideal parallel machine? On a CUDA GPU? How would you program it? What are the challenges?