Massive parallel programming on Graphics Processing Units and Applications (part 2)

Lokman Abbas-Turki
lokmane.abbas_turki@sorbonne-universite.fr

This project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No 101051997

October 2023

EUMaster4HPC

Plan

Install, notebook and documentation

- Install/use on Linux and Windows local machine
- Jupyter notebook and Google Colab
- Extensive documentation

GPU architecture and device query

- Runtime API and error handling
- GPU architecture
- GPU specifications

Hello World! and built-in variables

- Kernel and device functions
- “Hello World!” and fourth parallel granularity
- Different levels of synchronization

Addition of two arrays

- Main steps
- Indices, global memory and registers
- CPU timer vs. GPU timer

Real applications

- Monte Carlo simulation
- Partial Differential Equation simulation

EUMaster4HPC

Plan

Install, notebook and documentation

Install/use on Linux and Windows local machine
Jupyter notebook and Google Colab
Extensive documentation

GPU architecture and device query

Runtime API and error handling
GPU architecture
GPU specifications

Hello World! and built-in variables

Kernel and device functions
"Hello World!" and fourth parallel granularity
Different levels of synchronization

Addition of two arrays

Main steps
Indices, global memory and registers
CPU timer vs. GPU timer

Real applications

Monte Carlo simulation
Partial Differential Equation simulation

EUMaster4HPC

on Linux machines

Before Installation

- Check that you have a (Nvidia) GPU on which you can use CUDA: `lspci | grep -i nvidia`
- Verify Linux version: `uname -m && cat /etc/*release`

Installation steps

- gcc/g++ should be already available on your machine, otherwise install it
- Install CUDA: `https://developer.nvidia.com/cuda-downloads`
- Disabling Secure Boot on UEFI (BIOS)
- Add `/usr/local/cuda/bin` to `PATH` and `/usr/local/cuda/lib64` to `LD_LIBRARY_PATH`
- Any problem during installation or compilation? Check `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html` especially to deal with the conflicting installation

Compilation + Execution

- Compile `name.cu` using `nvcc name.cu -o nameex`
- The options `-arch=compute_Xx -code=sm_Yy` should be used sometimes
- For the compilation of various files, an example of a Makefile is also given
- Execute `nameex` using: `./nameex`

EUMaster4HPC

on Windows machines

Before Installation

- Check that you have a (Nvidia) GPU on which you can use CUDA: `control /name Microsoft.DeviceManager`
- Verify Windows version: `winver`

Installation steps

- Install Visual Studio 2022 Community with C/C++ tools: `https://visualstudio.microsoft.com/downloads/`
- Install CUDA: `https://developer.nvidia.com/cuda-downloads`
- Disabling Secure Boot on UEFI (BIOS)
- Add the address of `cl` compiler to `Path`
- Perform register changes explained at 7:40 in the video `https://www.youtube.com/watch?v=8NtHDkUoN98`
- Additional information on `https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html`

Compilation + Execution

- Compile `name.cu` using `nvcc name.cu -o nameex`
- The options `-arch=compute_Xx -code=sm_Yy` should be used sometimes
- For the compilation of various files, an example using CMake is also given
- Execute `nameex` using: `nameex`

EUMaster4HPC

Exercise session: local Jupyter or Google Colaboratory

Jupyter on a GPU CUDA-capable

- Perform the CUDA installation steps on either Linux or Windows machine
- Install Miniconda `https://docs.conda.io/projects/miniconda/en/latest/`
- Launch miniconda as administrator and create a virtual environment
- Activate the virtual environment then install everything you need, like: `conda install notebook` to install Jupyter notebook

Google Colab

- Have a Gmail account allows you to use Google Colab
- Open a new notebook
- Click on Runtime then change runtime type, choose T4 GPU
- To upload your Jupyter notebook, click on File.
- On the left toolbar, click on Files to upload your source code
- On your local machine, keep regular copies of the code written in Colab

Compilation + Execution

- Compile `name.cu` using `!nvcc name.cu -o nameex`
- Execute `nameex` using: `!./nameex`

EUMaster4HPC

Most Important documents!

Very often use the documentation provided by NVIDIA, in particular:

- **CUDA_C_Programming_Guide**: Document necessary for mastering CUDA API and overall understanding of the GPU hardware architecture
- **CUDA_Runtime_API**: Document describing functions from CUDA API

Specific documents

- **CUDA C++ Best Practices guide**: The third most important
- **CUDA_Math_API**: mathematical functions and number formats
- Other Math libraries: cuBLAS, cuFFT, cuRAND, cuSPARSE, and others
- Optimizations on each architecture, debug and profile a code, and others

CUDA for Python

Just-in-Time compilation proposed by Numba JIT functions is the best option `https://numba.pydata.org/numba-doc/latest/cuda/kernels.html`

Libraries using CUDA

- The best option for Machine Learning: `https://pytorch.org/docs/stable/notes/cuda.html#cuda-semantics`
- For general purpose data science: `https://docs.rapids.ai/user-guide`

Forums and code samples

- For specific questions, you can use `https://forums.developer.nvidia.com/c/accelerated-computing/cuda/206`
- Some CUDA code samples are installed with the CUDA toolkit: `https://developer.nvidia.com/cuda-code-samples`
- Alternative forum `https://stackoverflow.com/questions/tagged/cuda`

EUMaster4HPC

Plan

Install, notebook and documentation

Install/use on Linux and Windows local machine
Jupyter notebook and Google Colab
Extensive documentation

GPU architecture and device query

Runtime API and error handling
GPU architecture
GPU specifications

Hello World! and built-in variables

Kernel and device functions
"Hello World!" and fourth parallel granularity
Different levels of synchronization

Addition of two arrays

Main steps
Indices, global memory and registers
CPU timer vs. GPU timer

Real applications

Monte Carlo simulation
Partial Differential Equation simulation

EUMaster4HPC

Runtime API

- Leads to simpler code when compared to Driver API, but it also lacks the level of control that the Driver API has
- The name of functions is prefixed by cuda like `cudaGetDeviceCount` and `cudaGetDeviceProperties`
- Functions return `cudaError_t` enum type with

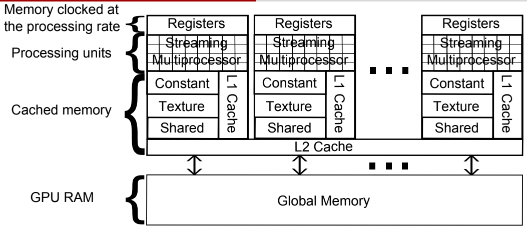
`cudaSuccess = 0`	The API call returned with no errors
`cudaErrorInvalidValue = 1`	Some parameters passed to the call are not within an acceptable range of values
...	
`cudaErrorUnknown = 999`	Unknown internal error has occurred

- But some values are deprecated like `cudaErrorInvalidDevicePointer = 17`

Error handling

```c
#include <stdio.h>
// Function that catches the error
void testCUDA(cudaError_t error, const char *file, int line) {
    if (error != cudaSuccess) {
       printf("There is an error in file %s at line %d\n",
              file, line);
       exit(EXIT_FAILURE);
    }
}
// Has to be defined in the compilation in order to get
// the correct value of the macros __FILE__ and __LINE__
#define testCUDA(error) (testCUDA(error, __FILE__ , __LINE__))
```

EUMaster4HPC



Multi-granularity parallelization

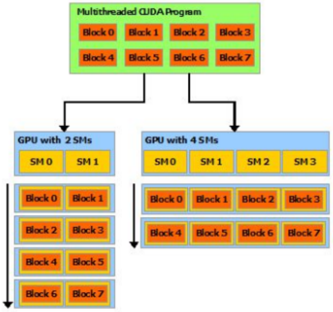
- Streaming processor: excutes threads
- Streaming multiprocessor: executes blocks
- Graphics Processing Cluster (new): executes grids

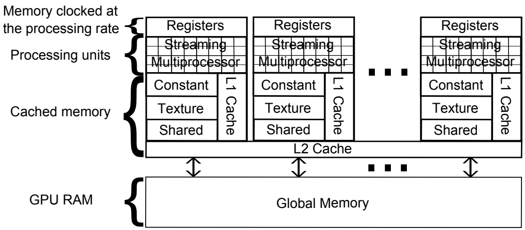
`empty_k<<<8,NTPB>>>();`

Depending on GPU capabilities

8 blocks of NTPB threads processed by either 2 SMs or 4 SMs

Figure from CUDA programming guide





GPU RAM

- contains global memory; values global to all threads
- could also contain local memory, values local to each thread

GPU cache

- Real cache: L1, L2 and shared memory
- Virtual cache: constant end texture memories

Main specifications Calling the function `cudaGetDeviceProperties (cudaDeviceProp *prop, int device)`

- `prop.totalGlobalMem, prop.sharedMemPerBlock, prop.regsPerBlock, prop.totalConstMem`
- `prop.major, prop.minor`
- `prop.maxThreadsPerBlock, prop.maxThreadsDim[3], prop.maxGridSize[3]`

EUMaster4HPC

Plan

Install, notebook and documentation

- Install/use on Linux and Windows local machine
- Jupyter notebook and Google Colab
- Extensive documentation

GPU architecture and device query

- Runtime API and error handling
- GPU architecture
- GPU specifications

Hello World! and built-in variables

- Kernel and device functions
- "Hello World!" and fourth parallel granularity
- Different levels of synchronization

Addition of two arrays

- Main steps
- Indices, global memory and registers
- CPU timer vs. GPU timer

Real applications

- Monte Carlo simulation
- Partial Differential Equation simulation

EUMaster4HPC

Function declaration and calling

Standard C functions

The same as for C or C++ programming

Kernel functions

- Called by the CPU and executed on the GPU
- Declared as `__global__ void myKernel (...) { ...; }`
- Called standardly by
  `myKernel<<<numBlocks, threadsPerBlock>>>(...);`
  where

  `numBlocks` should take into account the number of multiprocessors
  `threadsPerBlock` should take into account the warp size
- Dynamic parallelism: kernels can be called within kernels by the GPU and executed on the GPU

device functions

- Called by the GPU and executed on the GPU
- Declared as

```
__device__ void myDivFun (...)  { ...; }
__device__ float myDivFun (...)  { ...; }
```

- Called simply by `myDivFun(...)` but only within other device functions or kernels

EUMaster4HPC

Hello World!

- `int printf(const char *format[, arg, ...])` can be called within a kernel function or a device function
- `printf` returns the number of arguments parsed instead of the number of characters printed
- `printf` executed by the GPU involves both the CPU and the GPU, thus the CPU should wait for GPU results before continuing
- `cudaDeviceSynchronize` makes the host wait for compute device to finish
- The number of printed `Hello World!` depends on the number of threads that execute `printf("Hello World!")`

Built-in variables

Known within functions excuted on GPU: threadIdx.x, blockDim.x, blockIdx.x, gridDim.x

- `threadIdx.x` identifies each thread within its block
- `blockIdx.x` identifies each block within its grid
- `blockDim.x` is the size of each block in terms of number of threads
- `gridDim.x` is the size of each grid in terms of number of blocks
- Also .y and .z are possible but not used in this course
- The order of printing `blockIdx.x` varies
- The order of printing `threadIdx.x` varies less as they grouped within warps of 32 threads:
  First warp, `threadIdx.x` ordered from 0 to 31
  Second warp, `threadIdx.x` ordered from 32 to 63
  and so on

EUMaster4HPC

Synchronization of blocks

- Before compute capability 9.0, the NVIDIA Hopper GPU architecture, it is impossible to synchronize blocks without exiting the kernel function
- After compute capabilitty 9.0, it is possible to synchronize all the thread blocks in the same cluster, coscheduled on a single GPU Processing Cluster, using `cluster.sync()`

Synchronization of threads

Threads from the same block can be synchronized using `__syncthreads()`

Synchronization of warps

- Before compute capability 7.0, the NVIDIA Volta GPU architecture, warps used the same program counter shared amongst all 32 threads in the warp
- Before compute capability 7.0, no synchronization is needed between threads of the same warp
- For recent Nvidia architectures, of compute capability Y.y, one can force warp-synchronicity using the compilation option `-arch=compute_60 -code=sm_Yy` with Yy>60
- For recent Nvidia architectures, the synchronization of warps can be performed using `__syncwarp()`
- For any Nvidia architecture, long sequences of diverged execution by threads within the same warp should be avoided

EUMaster4HPC

Plan

Install, notebook and documentation

Install/use on Linux and Windows local machine
Jupyter notebook and Google Colab
Extensive documentation

GPU architecture and device query

Runtime API and error handling
GPU architecture
GPU specifications

Hello World! and built-in variables

Kernel and device functions
"Hello World!" and fourth parallel granularity
Different levels of synchronization

Addition of two arrays

Main steps
Indices, global memory and registers
CPU timer vs. GPU timer

Real applications

Monte Carlo simulation
Partial Differential Equation simulation

EUMaster4HPC

On CPU We want to add two large arrays of integers and put the result in a third one.

- Create a new file .cu, include `stdio.h` and `timer.h` in the top
- In the main function, allocate three arrays `a, b, c` using `malloc`
- Assign some values to `a` and `b`
- Using functions defined in `timer.h`, compute the execution time of adding `a` and `b`
- Free the CPU memory using `free`

On GPU We keep the same CPU code. For given values of *numBlocks* and *threadsPerBlock*, we want to perform an addition of *numBlocks* $\times$ *threadsPerBlock* integers

- Allocate `aGPU, bGPU, cGPU` on the GPU using `cudaMalloc`
- Transfer the values of `a, b` to `aGPU, bGPU` using `cudaMemcpy`
- Write the kernel that adds `aGPU` to `bGPU` and return the result in `cGPU`
- Copy `cGPU` to `c`
- Compute the execution time
- Free the GPU memory using `cudaFree`

EUMaster4HPC

Global index

- `int idx = threadIdx.x + blockIdx.x*blockDim.x;` ensures coalesced access to global memory
- `int idx = threadIdx.x*gridDim.x + blockIdx.x;` is however a very bad choice
- The value of each `idx` is stored in a register

Registers

- Divided homogeneously between threads of the same block
- Have a lifetime of a kernel
- Cannot be used for arrays

Unified Memory A single virtual address space accessible from the host and any device

- The double allocation using `malloc` and `cudaMalloc` can be replaced by `cudaMallocManaged`
- `cudaDeviceSynchronize` is needed to make the CPU wait for GPU before accessing the data
- Page table entries may not be created until they are accessed. The pages can migrate to any processor's memory at any time
- Unified memory is very beneficial when there is an important movement of data especially between the CPU and GPU
- Unified memory is not a good option when the data stay in the same memory space
- Unified memory is necessary for enabling peer-to-peer transfer of data, directly across the PCIe bus or NVLink, bypassing host memory

EUMaster4HPC

CPU timers
- Kernel launches and memory-copy functions are asynchronous, as are many CUDA API functions
- Necessary to synchronize by calling `cudaDeviceSynchronize` immediately before starting and stopping the CPU timer
- Used to time an overall solution that involves various kernel callings or that involves also the host

GPU timer
- Timing measures on the GPU and does not involve the operating system
- Best choice for local optimization, like the execution time of one kernel
- Timing is expressed in milliseconds

```
float TimeVar;
cudaEvent_t start, stop;
testCUDA(cudaEventCreate(&start));
testCUDA(cudaEventCreate(&stop));
testCUDA(cudaEventRecord(start,0));

/*********************************************************

To compute the execution time of this part of the code

*********************************************************/

testCUDA(cudaEventRecord(stop,0));
testCUDA(cudaEventSynchronize(stop));
testCUDA(cudaEventElapsedTime(&TimeVar, start, stop));
testCUDA(cudaEventDestroy(start));
testCUDA(cudaEventDestroy(stop));
```

EUMaster4HPC

Plan

Install, notebook and documentation

Install/use on Linux and Windows local machine
Jupyter notebook and Google Colab
Extensive documentation

GPU architecture and device query

Runtime API and error handling
GPU architecture
GPU specifications

Hello World! and built-in variables

Kernel and device functions
"Hello World!" and fourth parallel granularity
Different levels of synchronization

Addition of two arrays

Main steps
Indices, global memory and registers
CPU timer vs. GPU timer

Real applications

Monte Carlo simulation
Partial Differential Equation simulation

EUMaster4HPC

Pricing European $X = e^{-rT} f(S_T)$

$E(X) \approx \frac{X_1 + X_2 + ... + X_n}{n}$, using a family $\{X_i\}_{i \le n}$ of i.i.d $\sim X$

- **Strong law of large numbers**:

$$P\left(\lim_{n\to+\infty} \frac{X_1 + X_2 + ... + X_n}{n} = E(X)\right) = 1$$

- **Central limit theorem**: denoting $\epsilon_n = E(X) - \frac{X_1 + X_2 + ... + X_n}{n}$

$$\frac{\sqrt{n}}{\sigma}\epsilon_n \to G \sim \mathcal{N}(0,1)$$

- There is a 95% chance of having $|\epsilon_n| \le 1.96\frac{\sigma}{\sqrt{n}}$

Euler scheme for Black & Scholes model

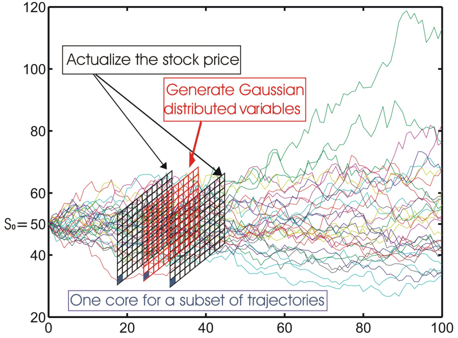
Given a time discretization sequence $t_k = kT/N$, with $k = 0, ..., N$ and $N = 100$, for $i = 1, ..., n$ we simulate

$$S^i_{t_k} = S^i_{t_{k-1}}\left[1 + rT/N + \sigma\sqrt{T/N}G^i_k\right], \quad S_0 = 50, \tag{1}$$

where the risk-free rate $r = 0.1$, the volatility $\sigma = 0.2$, and $(G^i_k)^{i=1,...,n}_{k=1,...,N}$ are independent normal random variables $\sim \mathcal{N}(0,1)$

Call Option

$f(x) = (x - K)_+ = \max(x - K, 0)$, with strike $K = S_0$ and maturity $T = 1$



For each time step $k < N$:

1 Random number generation (if parallelized) of G_k^i

2 Stock price actualization $S_{t_k}^i = S_{t_{k-1}}^i \left[1 + rT/N + \sigma\sqrt{T/N}G_k^i\right]$

Final time step $k = N$:

3 Compute the payoff $X^i = e^{-rT} f(S_T^i) = e^{-rT}(S_T^i - K)_+$

4 Send the values $(X^i)^{i=1,\dots,n}$ to the CPU for the approximation $E(X) \approx \dfrac{X_1 + X_2 + \dots + X_n}{n}$

EUMaster4HPC

General Form of linear RNGs

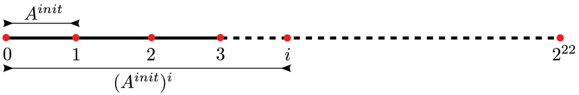
Without loss of generality:

$$X_n = (AX_{n-1} + C)\ mod(m) = (A\ :\ C)\begin{pmatrix} X_{n-1} \\ \dots \\ 1 \end{pmatrix}\ mod(m) \qquad (2)$$

Parallel-RNG from Period Splitting of One RNG

* Pierre L'Ecuyer proposed a very efficient RNG (1996) which is a CMRG on 32 bits: Combination of two MRG with $lag = 3$ for each MRG.
* Very long period $\sim 2^{185}$

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) mod(m)$$



Parallel-RNG from Parameterization of RNGs

* Same parallelization as SPRNG Prime Modulus LCG.
* The same RNG with different parameters "a":

$$x_n = a x_{n-1} + c \quad mod(m)$$

EUMaster4HPC

Black & Scholes PDE

$$\frac{\partial F}{\partial t}(t,x) + rx\frac{\partial F}{\partial x}(t,x) + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 F}{\partial x^2}(t,x) = rF(t,x), \quad F(T,x) = f(x)$$

With $u(t,x) = e^{r(T-t)}F(t,e^x)$, we equivalently solve the PDE

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t,x) + \mu\frac{\partial u}{\partial x}(t,x) = -\frac{\partial u}{\partial t}(t,x), \quad \mu = r - \frac{\sigma^2}{2}, \quad u(T,x) = f(e^x)$$

Put example

- $f(e^x) = \max(K - e^x, 0)$ where K is the strike
- The two limit conditions will be set at $x_{min} = \ln(K/2)$ and $x_{max} = \ln(2K)$ assuming heuristically that for all $t \in [0, T]$

$$u(t, x_{min}) = \mathtt{pmin} = K/2 \quad \& \quad u(t, x_{max}) = \mathtt{pmax} = 0 \tag{4}$$

PDE discretization

- σ takes its value in $[0.1, 0.5]$
- The volatility discretization involves $\mathtt{NB} = 64$ cells
- The space discretization involves $\mathtt{NTPB} = 256$ cells
- The time discretization of $[0, T]$ involves $\mathtt{N} = 10000$ time steps

EUMaster4HPC

PDE explicit discretization

$$u_{i,j} = p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1}, \;\; u_{i,j} = u(t_i, x_j), \;\; u(T,x) = f(e^x),$$
$$p_u = \frac{\sigma^2 \Delta t}{2\Delta x^2} + \frac{\mu \Delta t}{2\Delta x}, \quad p_m = 1 - \frac{\sigma^2 \Delta t}{\Delta x^2}, \quad p_d = \frac{\sigma^2 \Delta t}{2\Delta x^2} - \frac{\mu \Delta t}{2\Delta x}, \quad \mu = r - \frac{\sigma^2}{2}$$

PDE implicit discretization

$$u_{i+1,j} = q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1}, \quad u_{i,j} = u(t_i, x_j), \quad u(T,x) = f(e^x),$$
$$q_u = -\frac{\sigma^2 \Delta t}{2\Delta x^2} - \frac{\mu \Delta t}{2\Delta x}, \quad q_m = 1 + \frac{\sigma^2 \Delta t}{\Delta x^2}, \quad q_d = -\frac{\sigma^2 \Delta t}{2\Delta x^2} + \frac{\mu \Delta t}{2\Delta x}$$

Tridiagonal system

solved with Thomas algorithm

$$\begin{pmatrix} d_1 & c_1 & & & 0 \\ a_2 & d_2 & c_2 & & \\ & a_3 & d_3 & \ddots & \\ & & \ddots & \ddots & \ddots \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \qquad (7)$$

Forward phase

$$c'_1 = \frac{c_1}{d_1}, \; y'_1 = \frac{y_1}{d_1}, \; c'_i = \frac{c_i}{d_i - a_i c'_{i-1}}, \; y'_i = \frac{y_i - a_i y'_{i-1}}{d_i - a_i c'_{i-1}} \text{ when } i = 2, ..., n$$

Backward phase

$$x_n = y'_n, \; x_i = y'_i - c'_i x_{i+1} \text{ when } i = n-1, ..., 1$$

EUMaster4HPC

```
for(int k=0; k<N; k++){
   for(int j=0; j<NB; j++){
      sig = sigmin + dsig*j;
      mu = r - 0.5f*sig*sig;
      pu = 0.5f*(sig*sig*dt/(dx*dx) + mu*dt/dx);
      pm = 1.0f - sig*sig*dt/(dx*dx);
      pd = 0.5f*(sig*sig*dt/(dx*dx) - mu*dt/dx);
      for(int i=0; i<NTPB; i++){
         u = i+1;
         m = i;
         d = i-1;
         if(i==0){
           buff[i] = pmin;
         }else{
             if(i==NTPB-1){
               buff[i] = pmax;
             }else{
               buff[i] = pu*CPUTab[0][j][u] + pm*CPUTab[0][j][m] +
                         pd*CPUTab[0][j][d];
             }
         }
      }
      for(int i=0; i<NTPB; i++){
         CPUTab[0][j][i] = buff[i];
      }
   }
}
```