OpenMP and GPU Programming

GPU Intro

Emanuele Ruffaldi

https://github.com/eruffaldi/course_openmpgpu

PERCeptual RObotics Laboratory, TeCIP
Scuola Superiore Sant'Anna
Pisa,Italy

e.ruffaldi@sssup.it

April 12,2016

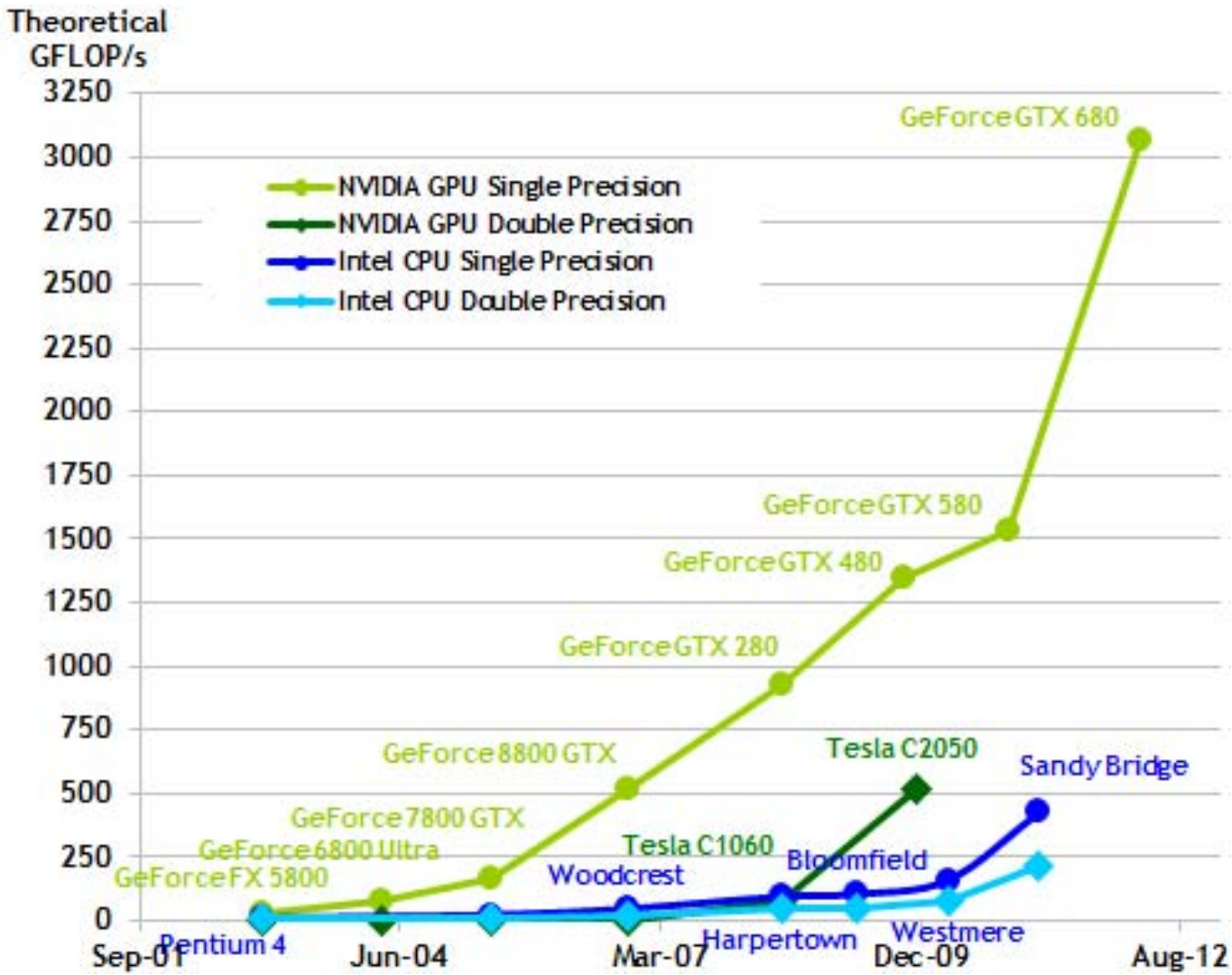
GPU

Graphic Processing Units (GPU) in the last decade have taken the lead in performance by exploiting specialized architectures with high parallelism.
This has allowed to move from pure graphic application to General Processing (GPGPU) providing new possibilities in the area of Machine Learning and Computer Vision running from in the cloud down to embedded GPUs in mobile, robotics and, more recently in cars.

- Which is the current performance level? GFlop/Watt, Dollar/Watt?
- Which architectural features have got GPU to such success?

GPU vs CPU

This is a graph showing the trend in GFLOP/s wrt the time.



Some Numbers

- Nowadays a NVidia GTX Titan X scores 6 TFlop in single precision, while the most powerful NVidia board is the NVidia DGX-1 providing 170 TFlops SP thanks to 8 GP100 GPUs (estimated 130k USD). Each Tesla P100 provides 11 TFlops SP with 15 Billion transistors.
- In comparison the last AMD GPU board is an Radeon Pro Duo providing 16 TFlops SP with 15-18 Billion transistors with two GPus.
- In comparison an Intel Core i7-6700K Processor has a theoretical throughput of 113 GFlops running at 4.2GHz with

NVidia Hardware

The building block is the Streaming Multiprocessor (SM) containing:

- cores organized in warps
- registers for cores
- number of threads running
- L1 cache
- shared memory for threads

Chips of the same NVidia generation differ in the number of SMs reported as the total number of cores, available memory and running frequency. For example the GTX Titan Black has 2880 cores.
NVidia hardware is organized in generations with different internal layout per-SM
Double precision requires additional cores inside each SM

NVidia Pascal SM

For example Pascal has each SM split in two sharing cache/shared memory, but not registers. The total number of cores is 64, much less than the 192/128 of previous Kepler/Maxwell.

GPU Origin and Working Principle

- A GPU is a heterogeneous chip multi-processor highly tuned for graphics
- Recall the approach of the graphic pipeline: vertex processing, per-pixel processing with large specialized memory access to textures
- Per-pixel processing is highly parallel but defined implicitly with specific order managed by the driver
- Example of units
  - Shader Core
  - Texture Unit (sampling)
  - Input Assembly
  - Rasterizer
  - Output Blend
  - Video Decoding (e.g. h.264)
  - Work Distributor

SIMD Working Principle

- Per-pixel processing follows the approach of Single Instruction Multiple Data (SIMD)
- In CPUs (e.g. x86 SSE, AVX) SIMD instructions are specified explicitly acting on 4 or 8 data elements
- In GPUs the vectorized execution is implicitly managed by the compiler while the developer specifies scalar instructions
- In particular NVidia uses a mix between flexibility and efficiency called Single Instruction Multiple Threads (SIMT)

In a **SIMT** system a group of parallel units (threads) are executed synchronously executing step by step the same instruction but impacting on different register contexts and on different local/global memory locations. Thanks to low-level thread indexing each unit performs the same task on a different part of the problem.

Explicit vs Implicit Vectorization

Example of explicit vectorization wit ARM Neon

```
void add(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
for(int i=0; i<n; i+=4) {
//compute c[i], c[i+1], c[i+2], c[i+3]
uint32x4_t a<4 = vld1q_u32(a+i);
uint32x4_t b4 = vld1q_u32(b+i);
uint32x4_t c4 = vaddq_u32(a4,b4);
vst1q_u32(c+i,c4);
}
}
```

CUDA scalar version

```
__global__ void add(float *a, float *b, float *c) {
int i = blockIdx.x * blockDim.x + threadIdx.x;
a[i]=b[i]+c[i]; //no loop!
}
```

Note also that CUDA supports float2 and float4.

Code taken from here.

NVidia SIMT Features

In an NVidia machine this group of synchronous threads is called a warp containing 32 threads (up to Maxell architecture). Warps are then allocated in a structure.

- Branching in NVidia SIMT is typically handled using instruction level predicates that mark if a single instruction is active due to some previous branching state
- Threads in the warp can communicate with three types of memory:
  - const memory
  - warp local memory
  - global memory
- The objective of the sync parallel execution is to achieve throughput by aligned memory access and shared dependencies

AMD Graphics Core Next

- GCN is the last iteration of AMD technology. It moved from a VLIW structure to RISC SIMD
- Conceptually similar to the NVIDIA SIMT approach
- Differently from the NVIDIA approach each GCN Compute Unit has 4 by 16 different low level units and it can deal with multiple instructions assigned at

CUDA Basics

CUDA is a toolkit that allows the development of NVidia GPU from the perspective of General Processing with GPUs. Here some terminology:

- Host: the computer
- Device: the GPU unit (more than one possible)

CUDA provides two approaches corresponding to two different API

- runtime: easy pre-compiled
- driver: complex, run-time compilation

The working principle of CUDA C/C++ compiler (nvcc) is a dual-path compilation: the C/C++ code contains special attributes that mark the fact that some code will be run on the Device, while the rest runs on the Host. The Host invokes the execution of the code on the Device with a novel syntax and the compiler hides the invocation of run-time functions that perform this operation.

CUDA Hello World

The tool nvcc takes regular C/C++ code (files with extension .cu) identifies the Device part, compiles both on CPU and on GPU and then assembles the executable in which the two elements are integrated. Take for example the following minimal hello world:

```
__global__ void kernel( void ) {
}
int main( void ) {
kernel<<<1,1>>>();
printf( "Hello, World!\n" );
return 0;
}
```

1. The kernel function is marked "__global__" to be executed on the Device and invoked by the CPU.
2. The triple bracket operator means to run the function kernel on the GPU with a parallelism specification, in this case it is executed once.

CUDA Summation

The previous example is a bit empty so we want to make it more practical by using the summation of elements. But before the execution we need to allocate memory.
Memory space of Host and Device are separate, so there is the need to allocate memory on GPU, transfer content back and forth the GPU before and after the execution.

- cudaMalloc, cudaFree,cudaMemcpy
- equivalent to malloc,free,memcpy, except that cudaMemcpy allows to specify the location of the transfer (Host/Device)

CUDA Summation - main

```
__global__ void add( int *a, int *b, int *c ) {
*c = *a + *b;
}
int main( void ) {
int a=2, b=7, c; // host copies of a, b, c
int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
int size = sizeof( int ); // we need space for an integer
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

CUDA Semantics of Kernel Invocation

The kernel invocation syntax allows to specify the 2D or 3D scheduling of the kernel execution, that depends on the semantics of the CUDA architecture:

```
kernel<<<blocks, threads>>>(args);
```

- The syntax allows to specify the number of blocks and the number of threads that make up a block
- In the code blocks are identified by blockIdx.x while threads by threadIdx.x
- The SIMT approach is based on the fact that every kernel execution in parallel accesses data that depends on the combination of blokcIdx and threadIdx.
- Inside the kernel it is possible to use blockDim. With the resulting indexing as follows

```
int index = threadIdx.x + blockIdx.x*blockDim.x
```

CUDA Why Threads

Blocks and Threads are a logical organization of the parallel task with the assumption that, much like Processes and Threads there is not easy memory sharing between Threads belonging to different Processes (Blocks). The scheduler maps Blocks/Threads to SM/Cores and in particular to SM/Warps: Blocks cannot span multiple SM meaning that there is a limited amount of possible Threads in a Block (typically 1024) and also shared memory is limited.
The case of dot product is very good for explaining the role of Threads

- Each thread computes the product part and result is stored in "shared" (block-level cache) memory
- The master thread of the block performs the summation
- Compare it against the SIMD approach

CUDA Why Threads (cont)

```
__global__ void dot( int *a, int *b, int *c ) {
__shared__ int temp[N];
temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
__syncthreads();
if( 0 == threadIdx.x ) {
int sum = 0;
for( int i = 0; i < N; i++ )
sum += temp[i];
*c = sum;
}
}
```

CUDA Dot Main

The main of the dot product operation invokes by obtaining one single result, and simply we could have one result per block. Alternatively we could have one single result by combining the results from each block BUT there is not guarantee so there the need of some form of atomic access.

CUDA invocation and SM

The block and thread invocation is a logical view that has to be mapped to physical threads running inside the SM. Shared memory is available per-SM meaning that logical threads of a block can be allocate only inside the same SM.

Limits

Know the limits of the invocation, in particular number of threads per block (512-1024 maximum)

Non Aligned ops

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

CUDA Memory Types

There are different types of memory in a CUDA application:

- Constant memory: the little values that remains constant across invocation (e.g. this corresponds to Uniforms in GLSL)
- Shared memory: the memory that threads of a block do share.
- Global memory: the memory allocated via cudaMalloc for performing the task
- Texture memory: a special type of memory, typically read-only, with interesting locality properties

CUDA Diagnostics

Classic printf can be called inside kernel and it is managed in a special (limited) buffer printed after kernel invocation.
Error printing is provided as follows:

```
{
    cudaError_t cudaerr = cudaPeekAtLastError();
    if (cudaerr != 0)
    printf("kernel launch failed with error \"%s\".\n",
    cudaGetErrorString(cudaerr));
}
```

CUDA CMake

CMake supports easily CUDA with the addition of new commands for invoking nvcc with the correct parameters. The following is a minimal usage:

```
find_package(CUDA QUIET REQUIRED)

cuda_add_executable(helloadd helloadd.cu)
cuda_add_executable(hellodot hellodot.cu)
cuda_add_executable(mandel mandel.cu)
```

CUDA Timing

Timing of operations is straightforward using an Event-based API.

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
```

CUDA OpenGL Interoperability (Extra)

OpenGL allows for hardware accelerated 3D graphics and it comes natural to think display GPGPU from CUDA to OpenGL or process OpenGL results in CUDA.
The solution is to map and unmap an OpenGL memory buffer into CUDA space.

- Assuming to have both a GPU and CUDA context
- Create OpenGL buffer
- Register the buffer in CUDA: cudaGLRegisterBufferObject
- Map the buffer to CUDA and use it: cudaGLMapBufferObject

CUDA Review

We can recap what we have understood from the CUDA toolkit:

- Tool: nvcc
- Attributes: __global__,__shared__
- Attributes: __host__,__device__
- Kernel Builtin Function: __syncthreads() printf()
- Kernel Builtin Variables: threadIdx,blockIdx,blockDim
- Builtin Functions: cudaMalloc cudaMemcpy cudaFree
- Builtin Type: dim3
- Invocation Syntax:

```
kernel<<<blocks, threads>>>(args);
```

```
dim3 blocks, threads;
kernel<<<blocks, threads>>>(args);
```

CUDA References

- Sanders, J. and Kandrot, E. CUDA by Example
- CUDA C Programming Guide
- CUDA C Best Practices Guide
- Programming Massively Parallel Processors: A Hands-on Approach, Kirk and Hwu, 2012, Morgan Kaufmann

CUDA Not covered topics

- Unified Memory
- Asynchrnous Invocation
- Occupancy calculation
- Multiple GPUs
- Dynamic Invocation

High-level Libraries and Tools for Heterogeneous CPU/GPU computing

CUDA and OpenCL provide the bulding blogs for the GPU computing with a reasonable level of abstraction, but something better can be done in particular to avoid vendor-lock-in.
We distinguish between custom language / directive approaches wrt API approaches. First the former:

- OpenMP 4.x+
- OpenACC

And then the APIs, in particular for C++:

- NVidia Thrust
- ViennaCL

We are interested in particular in the first one because it is a powerful C++ template library.

OpenCL

Open Computing Language is the standard-based alternative to CUDA. It is an independent C-like language with a corresponding API runtime. It provides many services for loading, parsing, compiling OpenCL and executing.
Each computer can have multiple OpenCL runtime by different vendors (e.g. Intel based on Multicore)
The syntax is more low-level than CUDA and due to the need of supporting multiple architecture (even FPGA) and it is somewhat more conservative, like in the sharing of pointers.
Some concepts in the C++ wrap of OpenCL:

- cl::Platform specifies the target platform
- cl::Context an access to a platform
- cl::Program a kernel
- cl::CommandQueue an entity of a platform where to put commands (memory transfer vs execution)
- cl::Buffer a slab of memory much like cudaMemalloc
- cl::KernelFunction a simplification of invocation

OpenMP 4.x+

- With OpenMP 4.0 it is possible to offload computations from the CPU to a Target accelerator being it a GPU or a custom external accelerator. The compilation principle is very similar to the one of CUDA but the offloading is provided via pragma directives.
- The master thread of CPU invokes the GPU (e.g. via CUDA) taking care of memory allocation and transfer
- The following nested pragmas allows the offloading:
  - omp target: specifies a GPU code
  - omp teams: controls the parallelism of the GPU code
  - omp distribute: specifies the behavior by block
  - omp parallel for: specifies the finer thread level action

OpenACC

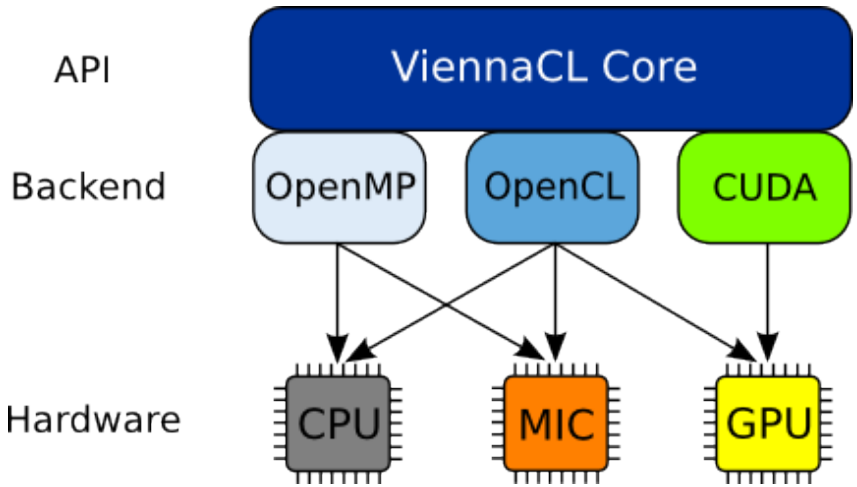
This standard takes a similar approach to OpenMP 4.0 by using "#pragma acc" with a mixture of C code and directives that specify kernels and loops with automatic data transfer.

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc kernels
    {
        #pragma acc loop
        for ( int j = 1; j < n-1; j++ ) {
            for ( int i = 1; i < m-1; i++ ) {
                Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                A [j-1] [i] + A [j+1] [i];
                error = fmax ( error, fabs (Anew [j] [i] - A [j] [i];
            }
        }

        #pragma acc loop
        for ( int j = 1; j < n-1; j++) {
            for (int = i; i < m-1; i++ ) {
                A [j] [i] = Anew [j] [i];
            }
        }
    }

    if (iter % 100 == 0) printf ("%5d, %0.6f\n", iter, error);
    iter++;
}
```

This standard is less common than OpenMP but it is still being updated and used in large paralell systems.

ViennaCL

ViennaCL (Linear Algebra) is a template C++ library specialized for matrix manipulation (also sparse) that was initially conceived to support CPU and OpenCL development and then it was ported to CUDA.
Internally it supports OpenCL, CUDA and OpenMP with switches that control the behavior.



Thrust

Thrust is a C++ template library that allows for high-level access to CUDA providing an easy to use interface to complex operations such as reductions and sort. The library is always up to date, made available with all CUDA SDKs.

Principle

- `thrust::host_vector<T>`
- `thrust::device_vector<T>`
- Also mapped from pure ptr:
  `thrust::device_ptr<int> dev_ptr(raw_ptr);`
- To ptr: `thrust::raw_pointer_cast(v);`

Requires some includes

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
```

Thrust Backends

While originally it was only CUDA based Thrust supports also CPU backends such as OpenMP and Intel TBB. They are selected via compilation time flags that typedef host_vector and device_vector.

Thrust Operations

All operations are based on C++ iterators with the possibility of using predicate or functors implemented using device-enabled functions.

Basic Operations

- Creation with constant value
- `thrust::fill(b,e,value)`
- `thrust::sequence(b,e)`
- `thrust::copy(b,e,d)`
- `thrust::replace(b,e,bad,good)`

Thrust Transformations

Transformations of data allows to perform basic unary and binary operations:

- `thrust::transform(b,e,d,thrust::negate<int>());`

Various operators are available and they can be customized as follows:

```
struct saxpy_functor {
    const float a;
    saxpy_functor(float _a) :
        a(_a) {}
    __host__ __device__ float operator()(const float& x, const float& y) const
    {
        return a * x + y;
    }
};
```

Thrust Reduction

As discussed in the general CUDA discussion reductions are complex operations but in Thrust they are quite simple and decomposed in the memberwise-operation (*) and reduction operation (+):

```
std::sqrt( thrust::transform_reduce(
    b,e,
    unary_op, init, binary_op) )
```

Note

The functional flexibility of reduction allows to make a single-pass minmax.

Thrust Conclusions

Other operations are possible with sort and union, for example.
Examples here
Limitations:

- Focused on 1D algorithms
- Learning curve