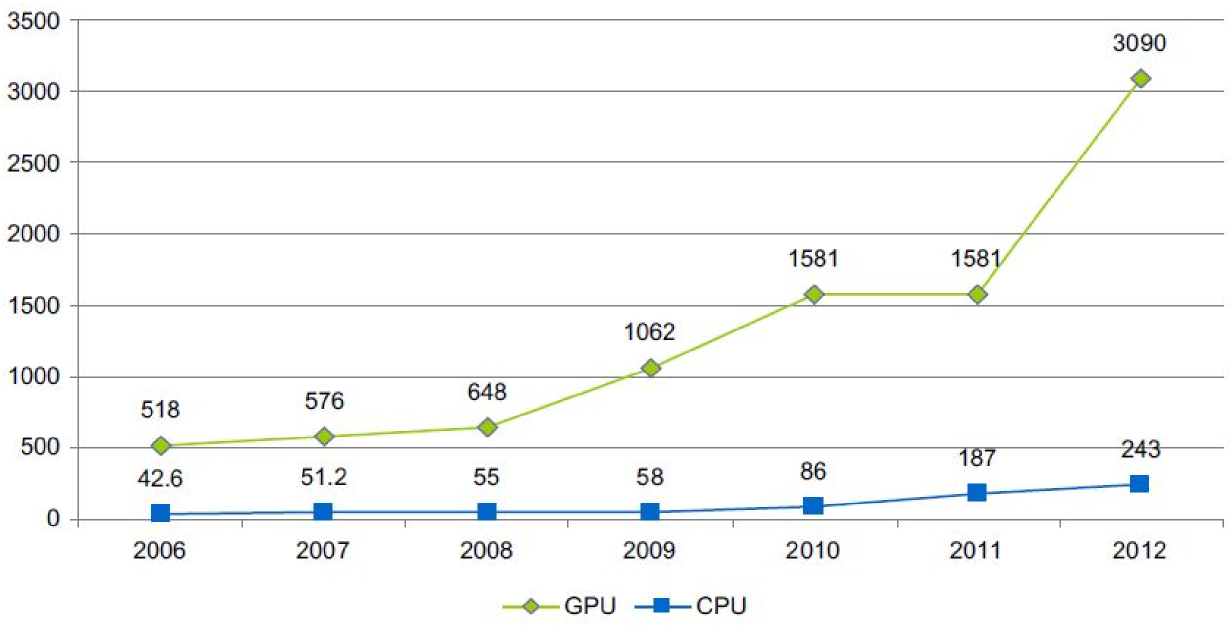
Introduction to the GPU Platform

András Wirth

GPU History

1993 – Nvidia Co founded
1994 – 3dfx Interactive founded
1995 – first chip NV1 introduced by Nvidia
1996 – 3dfx released Voodoo Graphics
1999 – GeForce 256 from Nvidia offered geometrical transformations support
2000 – Nvidia acquires 3dfx Interactive
2002 – GeForce 4 equipped with pixel and vertex shaders
2006 – GeForce 8 – unified computing architecture (not distinguishing pixels and vertex shaders) – Nvidia CUDA
2008 – GeForce 280 – supports computing in double FPU precision
2010 – GeForce 480 (Fermi) – first GPU designed directly for general purpose GPU computing

GPU History



3500
3000
2500
2000
1500
1000
500
0
3090
1581
1581
1062
518
576
648
243
187
86
42.6
51.2
55
58
2006
2007
2008
2009
2010
2011
2012
GPU
CPU

CPU vs GPU

CPU vs GPU

	Nvidia GeForce 580	Intel i7-960 6xCore
Transistors	3000 * 10^6	1170 * 10^6
Frequencty	1.5 GHz	3.5 GHz
Num. of threads	512	12
Performance	1.77 Tflops	~200 GFLops
Throughput	194 GB/s	26 GB/s
RAM	1.5 GB	~48GB
Load	244W	130W

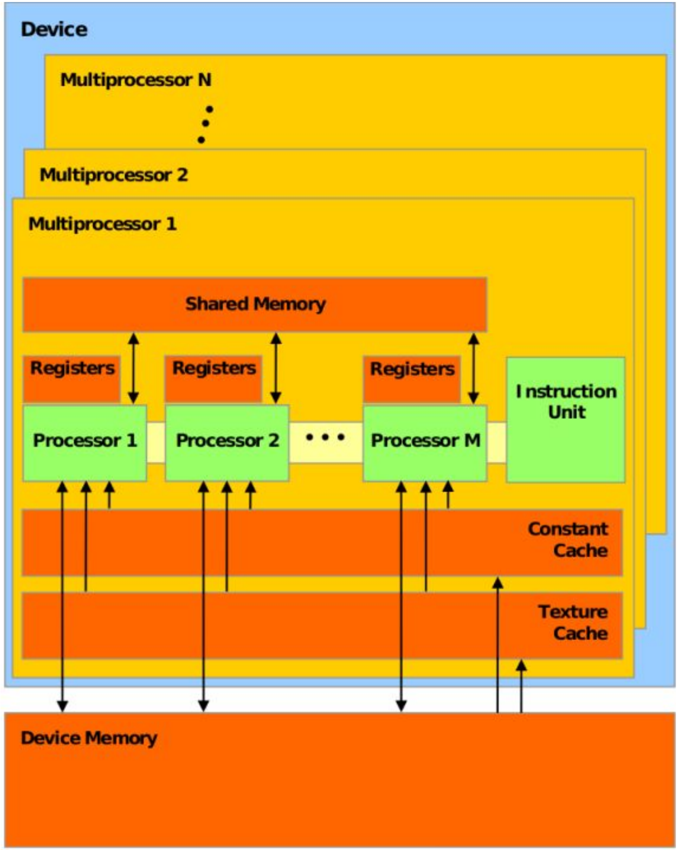
Cache/Memory Latency Comparison

	L1	L2	L3	Main Memory
AMD FX-8150 (3.6GHz)	4	21	65	195
AMD Phenom II X4 975 BE (3.6GHz)	3	15	59	182
AMD Phenom II X6 1100T (3.3GHz)	3	14	55	157
Intel Core i5 2500K (3.3GHz)	4	11	25	148
Intel Core i7 3960X (3.3GHz)	4	11	30	167

Memory Bandwidth Comparison - Sandra 2012.01.18.10

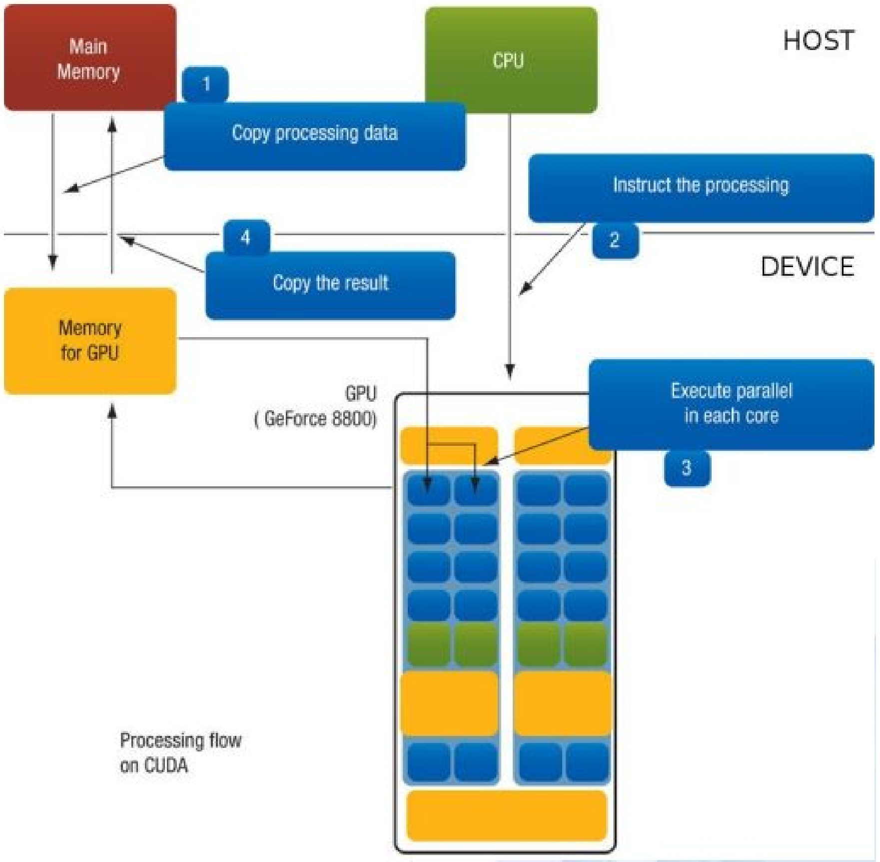
	Intel Core i7 3960X (Quad Channel, DDR3-1600)	Intel Core i7 2600K (Dual Channel, DDR3-1600)	Intel Core i7 990X (Triple Channel, DDR3-1333)
Aggregate Memory Bandwidth	37.0 GB/s	21.2 GB/s	19.9 GB/s

CUDA Architecture

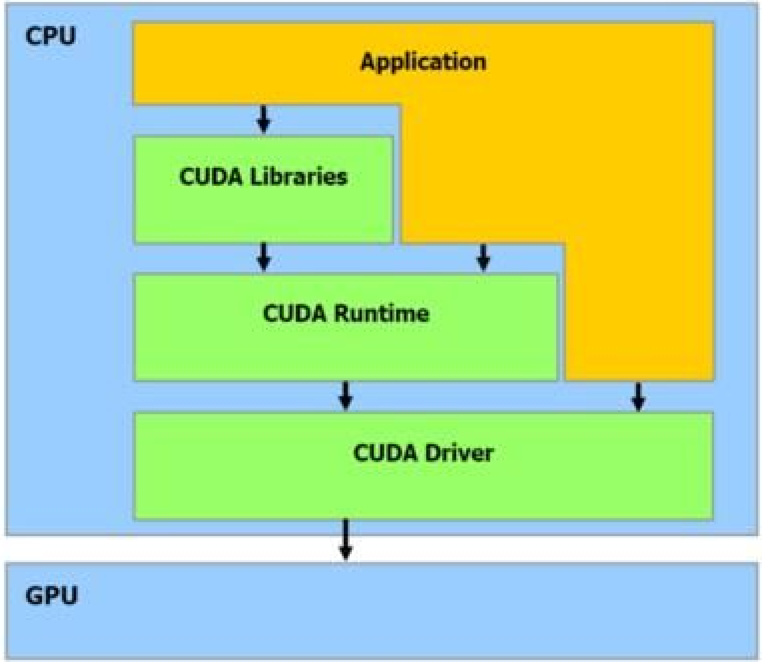


Computing Process

1. Copy data from the HOST memory to the DEVICE memory.
2. Start threads in DEVICE
3. Execute threads in GPUs multiprocessors.
4. Copy results back from the DEVICE memory to the HOST memory



CUDA Runtime vs. Driver API

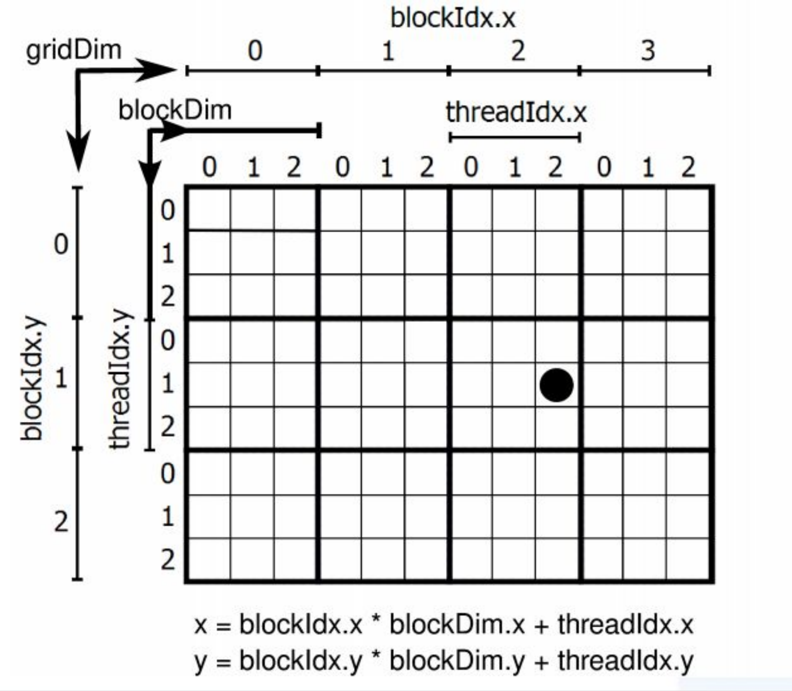


- Driver API: low-level, more flexible, harder to develop (similar to OpenCL)
- Runtime API: higher level, more convenient, faster development, easier debugging

CUDA Programming

- The kernel is a function for GPU threads. The C/C++ is extended for kernel function execution by command “name<<<...>>>(...)”.
- __device__ is a function modifier. This function will be executed in the device and it can be called only from the device.
- __host__ is opposite function modifier than __device__. Functions marked with this modifier are only for the CPU.
- __global__ is modifier for kernels. Function will be executed in GPU, but called (started) is from CPU.

CUDA Grid Organization



CUDA Programming

CUDA C/C++ Kernel Code:

```
__global__ void
vectorAdd(const float * a, const float * b, float * c)
{
    // Vector element index
    int nIndex = blockIdx.x * blockDim.x + threadIdx.x;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL Kernel Code

```
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global       float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

CUDA Programming

CUDA Driver API

```
// Kernel launch configuration
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks    = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

CUdevice    hDevice;
CUcontext   hContext;
CUmodule    hModule;
CUfunction  hFunction;

// create CUDA device & context, and load the kernel
cuInit(0);
cuDeviceGet(&hContext, 0); // pick first device
cuCtxCreate(&hContext, 0, hDevice));
cuModuleLoad(&hModule, “vectorAdd.cubin”);
cuModuleGetFunction(&hFunction, hModule, "vectorAdd");

 // allocate host vectors
 float * pA = new float[cnDimension];
 float * pB = new float[cnDimension];
 float * pC = new float[cnDimension];

 // initialize host memory (using helper C function called “r
 randomInit(pA, cnDimension);
 randomInit(pB, cnDimension);
```

OpenCL

```
// Kernel launch configuration
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks    = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

// Get OpenCL platform count
cl_uint NumPlatforms;
clGetPlatformIDs (0, NULL, &NumPlatforms);

// Get all OpenCL platform IDs
cl_platform_id* PlatformIDs;
PlatformIDs = new cl_platform_id[NumPlatforms];
clGetPlatformIDs(NumPlatforms, PlatformIDs, NULL);

// Select NVIDIA platform (this example assumes it IS present)
char cBuffer[1024];
cl_uint NvPlatform;
for(cl_uint i = 0; i < NumPlatforms; ++i)
{
  clGetPlatformInfo (PlatformIDs[i], CL_PLATFORM_NAME, 1024, cBuf
  if(strstr(cBuffer, "NVIDIA") != NULL)
  {
    NvPlatform = i;
    break;
  }
}
```

CUDA Programming

```
// allocate memory on the device
CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;
cuMemAlloc(&pDeviceMemA, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemB, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemC, cnDimension * sizeof(float));

// copy host vectors to device
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension * sizeof(float));
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension * sizeof(float));

// setup parameter values
cuFuncSetBlockShape(hFunction, cnBlockSize, 1, 1);
cuParamSeti(hFunction, 0, pDeviceMemA);
cuParamSeti(hFunction, 4, pDeviceMemB);
cuParamSeti(hFunction, 8, pDeviceMemC);
cuParamSetSize(hFunction, 12);

// execute kernel
cuLaunchGrid(hFunction, cnBlocks, 1);

// copy the result from device back to host
cuMemcpyDtoH((void *) pC, pDeviceMemC, cnDimension * sizeof(fl

// cleanup
delete[] pA;
delete[] pB;
delete[] pC;
cuMemFree(pDeviceMemA);
cuMemFree(pDeviceMemB);
cuMemFree(pDeviceMemC);
```

```
// Get a GPU device on Platform (this example assumes one IS prese
cl_device_id cdDevice;
clGetDeviceIDs(PlatformIDs[NvPlatform], CL_DEVICE_TYPE_GPU, 1,
               &cdDevice, NULL);

// Create a context
cl_context hContext;
hContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, NULL);

// Create a command queue for the device in the context
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, cdDevice, 0, NULL);

// Create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource,
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// Create kernel instance
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd", 0);

// Allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];
```

CUDA Programming

```
// Initialize host memory (using helper C function called "randomInit")
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);

// Allocate device memory (and init hDeviceMemA and hDeviceMemB)
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float), pB, 0);
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             cnDimension * sizeof(cl_float), 0, 0);
// Setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);

// Launch kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// Copy results from device back to host;  block until complete
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                    cnDimension * sizeof(cl_float), pC, 0, 0, 0);
```

CUDA Programming

```
// Cleanup
delete[] pA;
delete[] pB;
delete[] pC;
delete[] PlatformIDs;
clReleaseKernel(hKernel);
clReleaseProgram(hProgram);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

CUDA Programming

CUDA Runtime API

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Optimization Considerations

- Minimize data transfer between host and device. In ideal case transfer data only twice. Before and after computing.
- Use GPU only for task with very intensive calculations.
- GPU with shared memory on the board would be more suitable.
- For intensive data transfer between CPU-GPU use pipelining.
- GPU computing can be used alongside data transfer GPU-CPU or CPU computing.
- Optimize access to shared memory. Sequential access is much faster than random access.
- Reduce divergent threads.
- Select optimal thread grid.