CUDA Programming Assist Tool Development

Postech Graphics Lab
이 승 용

Contents

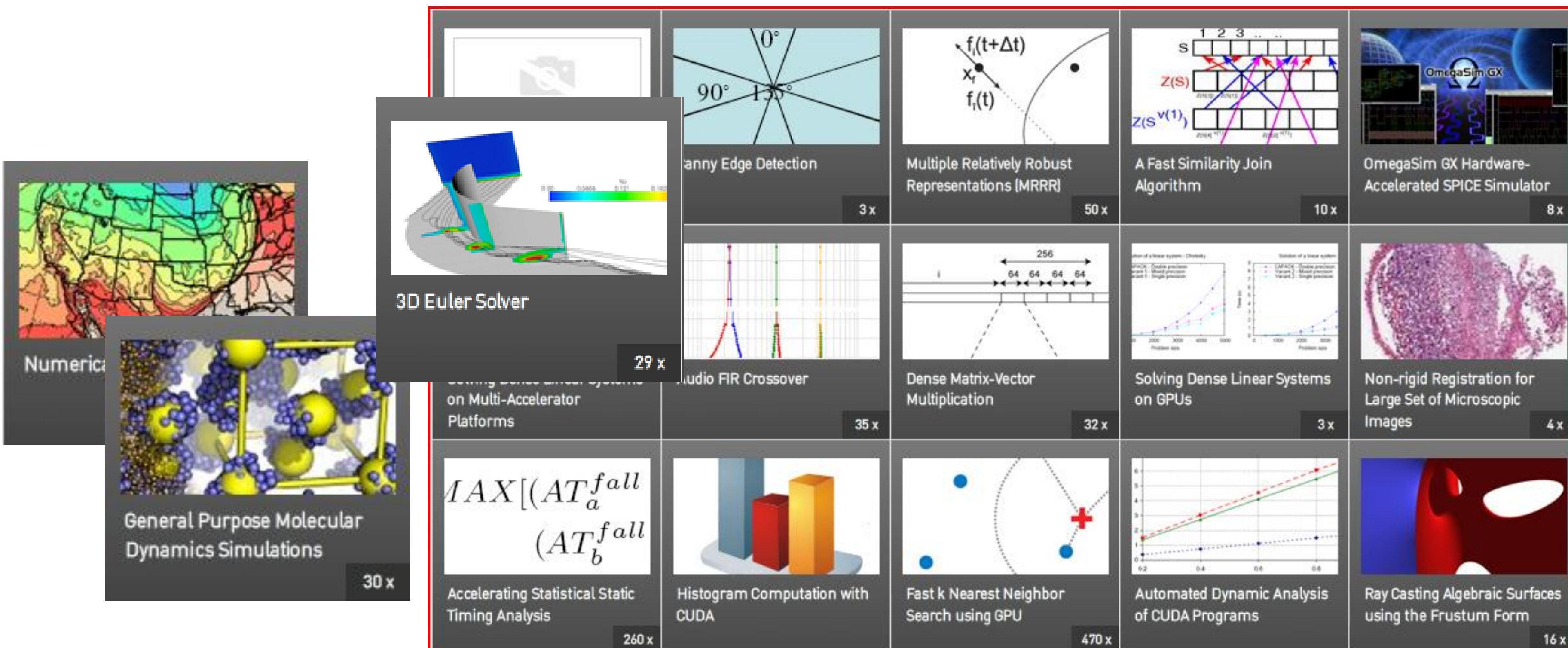
- GPU
- GPGPU
- CUDA
- Difficulties in CUDA Programming
- CUDA Programming Assist Tool

GPU?

- A massively parallel programmable processor
- Present in almost every PC
- Have large amount of arithmetic capability (especially floating-point operation)

GPGPU !

- Various application with heavy computation
  - Accelerated using computational power of GPU
  - Expanded for general purpose



GPGPU Programming

- Goal
  - Increase the overall throughput of the computer system on the given task
  - Use CPU and GPU synergistically

- Difficulties:
  - programmer should know about traditional graphics rendering pipeline (GLSL ,HLSL, Cg)

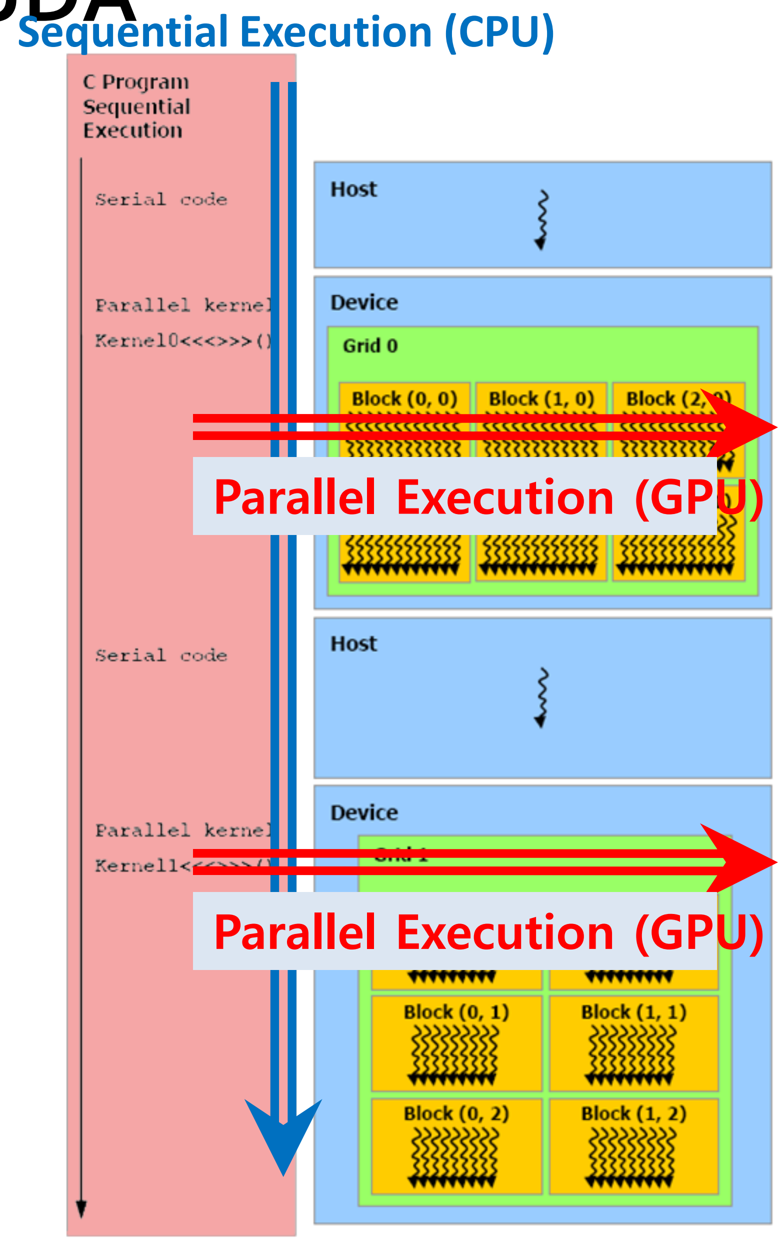
CUDA- Compute Unified Device Architecture

- Co-designed HW and SW to expose the computational horsepower of nVIDIA GPUs for GPU computing
- Most popular GPGPU language
- C language with minimal extension
  - easy to learn
  - more similar to ordinary programming

Parallel Computing with CUDA

- By specifying the number of parallel kernels, we can get data parallelism easily

- SIMT – all threads execute the same kernel code with different data



Serial code executes on the host while parallel code executes on the device.

General CUDA Programming

- In HOST program
  - Device management
  - Memory allocation & deallocation
  - Memory Copy (CPU->GPU, GPU->CPU)
  - Thread launch
  - Synchronization

- In DEVICE program
  - Actual kernel function.

Difficulties in CUDA Programming

- HOST program parts are error-prone, but not essential parts of actual parallel codes.
- Writing these repeated codes is a tedious, boring task.
- Effective tool support needed.
  -> frame based programming (XVCL)

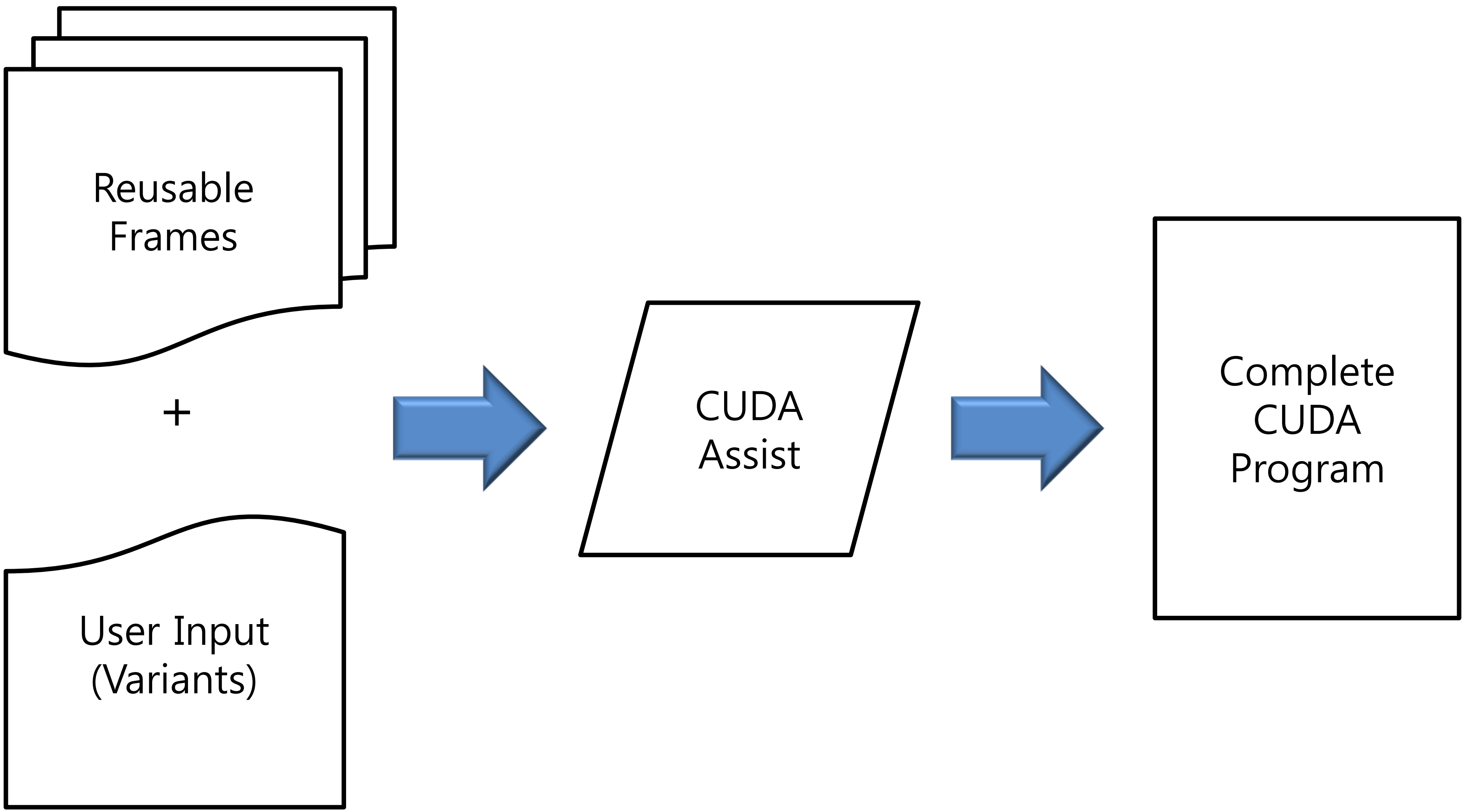
```
void initDenoiseCUDA( int imageh, int imagew)
{

    img_size_H = imageh;
    img_size_W = imagew;

    std::cout << "Allocate GPU memory..." << std::endl;

    cutilSafeCall( cudaMalloc((void **)&d_imgL, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_imgR, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_dstR, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_imga, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_imgG, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_dstG, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_imgb, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_imgB, img_size_H * img_size_W * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_dstB, img_size_H * img_size_W * sizeof(float)) );


    unsigned int freeGPUmem, totalGPUmem;       // for checking the GPU memory
    int ret = cuMemGetInfo(&freeGPUmem, &totalGPUmem);
    if (ret != CUDA_SUCCESS) {
        std::cout << "cuMemGetInfo failed! [status = " << ret << "]" << std::endl;
        return;
    }
    else {
        std::cout << "GPU Memory total: " << totalGPUmem << ", free: " << freeGPUmem << std::endl;
    }
}


void bilateralCUDA(float *h_dstR,float *h_dstG,float *h_dstB,
                   float *h_imgR,float *h_imgG,float *h_imgB,
                   float *h_imgL,float *h_imga,float *h_imgb,
                   float *h_gaussian,
                   float sigma_s,float sigma_r)
{
    cutilSafeCall( cudaMemcpy(d_imgR, h_imgR, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi
    cutilSafeCall( cudaMemcpy(d_imgG, h_imgG, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi
    cutilSafeCall( cudaMemcpy(d_imgB, h_imgB, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi
    cutilSafeCall( cudaMemcpy(d_imgL, h_imgL, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi
    cutilSafeCall( cudaMemcpy(d_imga, h_imga, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi
    cutilSafeCall( cudaMemcpy(d_imgb, h_imgb, img_size_H * img_size_W * sizeof(float), cudaMemcpyHostToDevi


    //templateCUDA();
    f_radius = (int)(sigma_s * 3.0f);
    f_diameter = f_radius*2 + 1;
    cutilSafeCall( cudaMalloc((void **)&d_gaussian, f_diameter * f_diameter * sizeof(float)) );
    cutilSafeCall( cudaMemcpy(d_gaussian, h_gaussian,f_diameter*f_diameter*sizeof(float), cudaMemcpyHostToD
    bilateral_CUDA(f_diameter, sigma_r);
    cutilSafeCall( cudaThreadSynchronize() );
    cutilSafeCall( cudaFree(d_gaussian) );


    cutilSafeCall( cudaMemcpy(h_dstR, d_dstR, img_size_H * img_size_W * sizeof(float), cudaMemcpyDeviceToHo
    cutilSafeCall( cudaMemcpy(h_dstG, d_dstG, img_size_H * img_size_W * sizeof(float), cudaMemcpyDeviceToHo
    cutilSafeCall( cudaMemcpy(h_dstB, d_dstB, img_size_H * img_size_W * sizeof(float), cudaMemcpyDeviceToHo
}
```

XVCL- xml based variant configuration language

- General-purpose mark-up language for configuring variants in a variety of software assets.
- Can be used for managing variants in any collection of textual documents so that SW reusability will increase.
- XVCL processor traverses related x-frames, interprets XVCL commands, and assembles the output (a custom program) into one or more files.

  (x-frame is an XML file with program code + XVCL commands)

CUDA Assistant Tool using XVCL



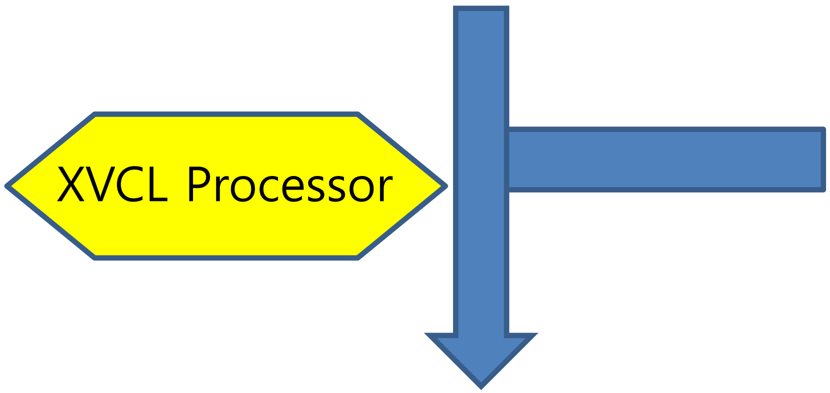
CUDA Assistant Tool using XVCL

- User can specify thread dimension, block dimension, data array variables, and kernel function as variant features.

- Reusable frames, such as memory management, thread launch, can be combined with variant features to make complete CUDA program.

```
x-frame SPC
x-frame SPC<set funcName = initDenoiseCUDA/>
<set arguments = int imageh, int imagew />
<set data_array = R, G, B, L, a, b />
<set array_size = imageh*imagew />
<set data_type  = float />
<adapt initCUDA />
```

```
x-frame initCUDA
void @funcName
( <while arguments>
   @arguments
  </while>
)
{
     std::cout << "Allocate GPU memory..." << std::endl;
     <while data_array>
     cutilSafeCall( cudaMalloc((void **)& @data_array ,
      @array_size * sizeof(@data_type)) );
     </while>
}
```




```
void initDenoiseCUDA(int imageh, int imagew)
{
		std::cout << "Allocate GPU memory..." << std::endl;

		cutilSafeCall( cudaMalloc((void **)&R, imageh*imagew * sizeof(float)) );
		cutilSafeCall( cudaMalloc((void **)&G, imageh*imagew * sizeof(float)) );
		cutilSafeCall( cudaMalloc((void **)&B, imageh*imagew * sizeof(float)) );
		cutilSafeCall( cudaMalloc((void **)&L, imageh*imagew * sizeof(float)) );
		cutilSafeCall( cudaMalloc((void **)&a, imageh*imagew * sizeof(float)) );
		cutilSafeCall( cudaMalloc((void **)&b, imageh*imagew * sizeof(float)) );
}
```

CUDA Assistant Tool Implementation

- Reusable frame
  - device initialization
  - memory allocation/deallocation
  - memory copy (CPU->GPU, GPU->CPU)
  - thread launch
- Variant
  - number of threads, blocks
  - data type
  - data array names
  - actual kernel function

Advantage of CUDA Assist Tool

- Easy memory management
  - will reduce error-prone coding
  - will increase productivity

- Extract parallelism as an independent concern and program it explicitly
  - pluggable (or attachable) parallelism
  - will allow effective management and reuse of CUDA programs

Future Direction

- High-level programming assist tool for CUDA
  - BSGP (Bulk-Synchronous GPU Programming)
  - Productive development of error-free SWs
- Programming assist tool for massive parallelism
  - Hundreds of threds
  - Shared and local memory

Thank you!

http://cg.postech.ac.kr