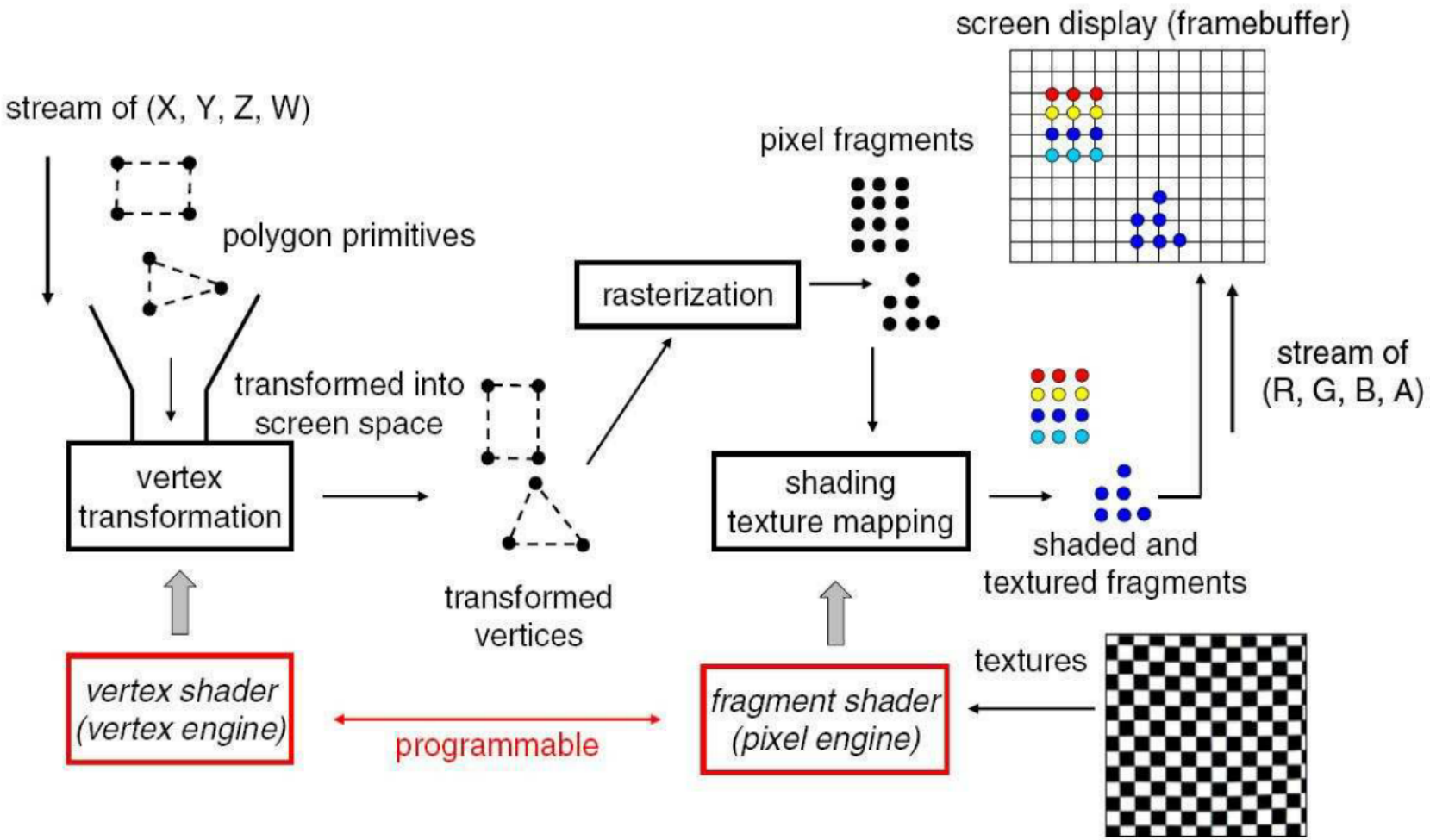
GPGPU Course

Introduction to GPGPU programming

Guillaume Raffy (guillaume.raffy@univ-rennes1.fr)

October 10, 2017

GPUs before CUDA



HLSL example : Phong pixel shader

phong.hlsl

```
struct VertexShaderOutput
{
  float4 Position : POSITION0;
  float2 TexCoords : TEXCOORD0;
  float3 Normal : TEXCOORD1;
  float3 WorldPos : TEXCOORD2;
};

float4 PixelShader(VertexShaderOutput input) : COLOR0
{
  // Phong relfection is ambient + light-diffuse + spec highlights.
  // I = Ia*ka*Oda + fatt*Ip[kd*Od(N.L) + ks(R.V)^n]
  // Get light direction for this fragment
  float3 lightDir = normalize(input.WorldPos - LightPosition);
  float diffuseLighting = saturate(dot(input.Normal, -lightDir)); // per pixel >
       >diffuse lighting

  // Introduce fall-off of light intensity
  diffuseLighting *= (LightDistanceSquared / dot(LightPosition - input.WorldPos, >
       >LightPosition - input.WorldPos));

  // Using Blinn half angle modification for perofrmance over correctness
  float3 h = normalize(normalize(CameraPos - input.WorldPos) - lightDir);

  float specLighting = pow(saturate(dot(h, input.Normal)), SpecularPower);

  return float4(saturate(
    AmbientLightColor +
    (DiffuseColor * LightDiffuseColor * diffuseLighting * 0.6) + // Use light >
         >diffuse vector as intensity multiplier
    (SpecularColor * LightSpecularColor * specLighting * 0.5) // Use light specular >
         >vector as intensity multiplier
    ), 1);
}
```

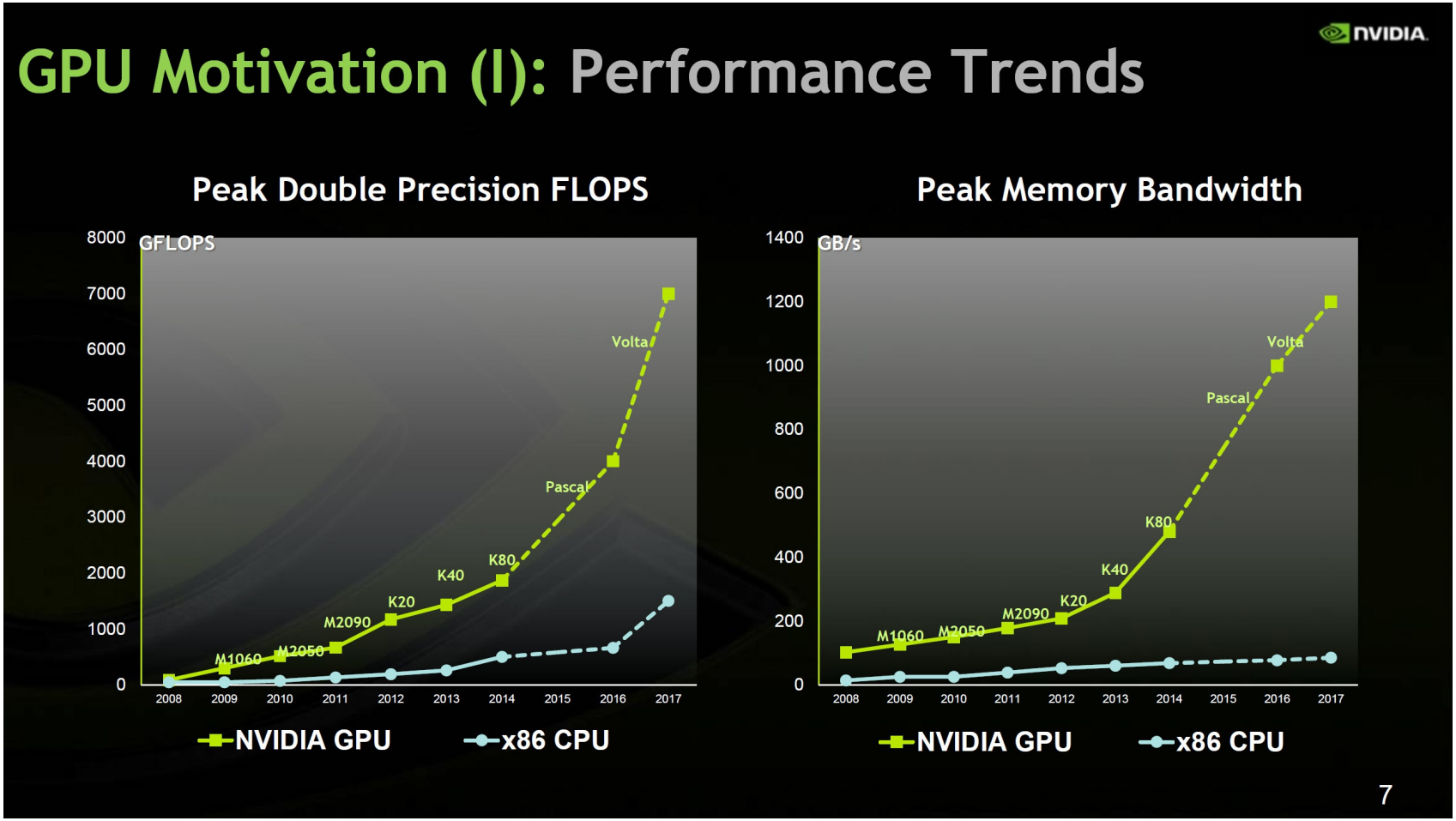
NVIDIA GPU Architecture evolution 1/2

Board name	C870	C1060	C2050	K20X
GPU name		T10	GF100	GK110
Arch name	G80	Tesla	Fermi	Kepler
release year	2006	2008	2010	2012
Fabrication process	90 nm	65 nm	40 nm	28nm
number of transistors	681 10^6	1400 10^6	3000 10^6	7080 10^6
number of SM	8	30	16 SM	14 SMX
number of cores per SM	16	8	32	192
number of cores	128	240	512	2688
TDP	170.9 *W*	187.8 *W*	238 *W*	235 *W*
SP Peak GFLOPs	518.4	933.12	1030	3950
DP Peak GFLOPs		78	515	1310
GPU size			529 mm^2	561 mm^2
Memory size	1536 MB	4000 MB	3072 MB	6144 MB
Memory bandwith	76.8 GB/s	102 GB/s	144 GB/s	250 GB/s
CUDA Compute capabilities	1.0	1.3	2.0	3.5

NVIDIA GPU Architecture evolution 2/2

Board name	K40	P100	V100 (PCI)
GPU name	GK110B	GP100	GV100
Arch name	Kepler	Pascal	Volta
release year	2012	2016	2017
Fabrication process	28nm	16nm	12nm
number of transistors	7080 10^6	15300 10^6	21100 10^6
number of SM	15 SMX	56	80
number of cores per SM	192	64	64
number of cores	2880	3584	5120
TDP	235 W	250 W	250 W
SP Peak GFLOPs	4291	9519	14028
DP Peak GFLOPs	1430	4760	7014
GPU size	561 mm^2	610 mm^2	815 mm^2
Memory size	12288 MB	16384 MB	16384 MB
Memory bandwith	288 GB/s	720 GB/s	900 GB/s
CUDA Compute capabilities	3.5	6.0	7.0

Peak GFLOPs evolution



CPU peak performance

Intel Xeon Gold 6140 (released in 2017)

- 2 CPUs per motherboard
- 18 cores per CPU
- 2 AVX-512 (512 bits wide SIMD) FMA units per core
- clock speed : 2.3 GHz

Theoretical double precision peak performance :

- number of double precision numbers per SIMD register :

CPU peak performance

Intel Xeon Gold 6140 (released in 2017)

- 2 CPUs per motherboard
- 18 cores per CPU
- 2 AVX-512 (512 bits wide SIMD) FMA units per core
- clock speed : 2.3 GHz

Theoretical double precision peak performance :

- number of double precision numbers per SIMD register : 8 (512/64)
- number of double precision operations per core per cycle :

CPU peak performance

Intel Xeon Gold 6140 (released in 2017)

- 2 CPUs per motherboard
- 18 cores per CPU
- 2 AVX-512 (512 bits wide SIMD) FMA units per core
- clock speed : 2.3 GHz

Theoretical double precision peak performance :

- number of double precision numbers per SIMD register : 8 (512/64)
- number of double precision operations per core per cycle : 32 ($8 \times 2 \times 2$)
- double precision gflops :

CPU peak performance

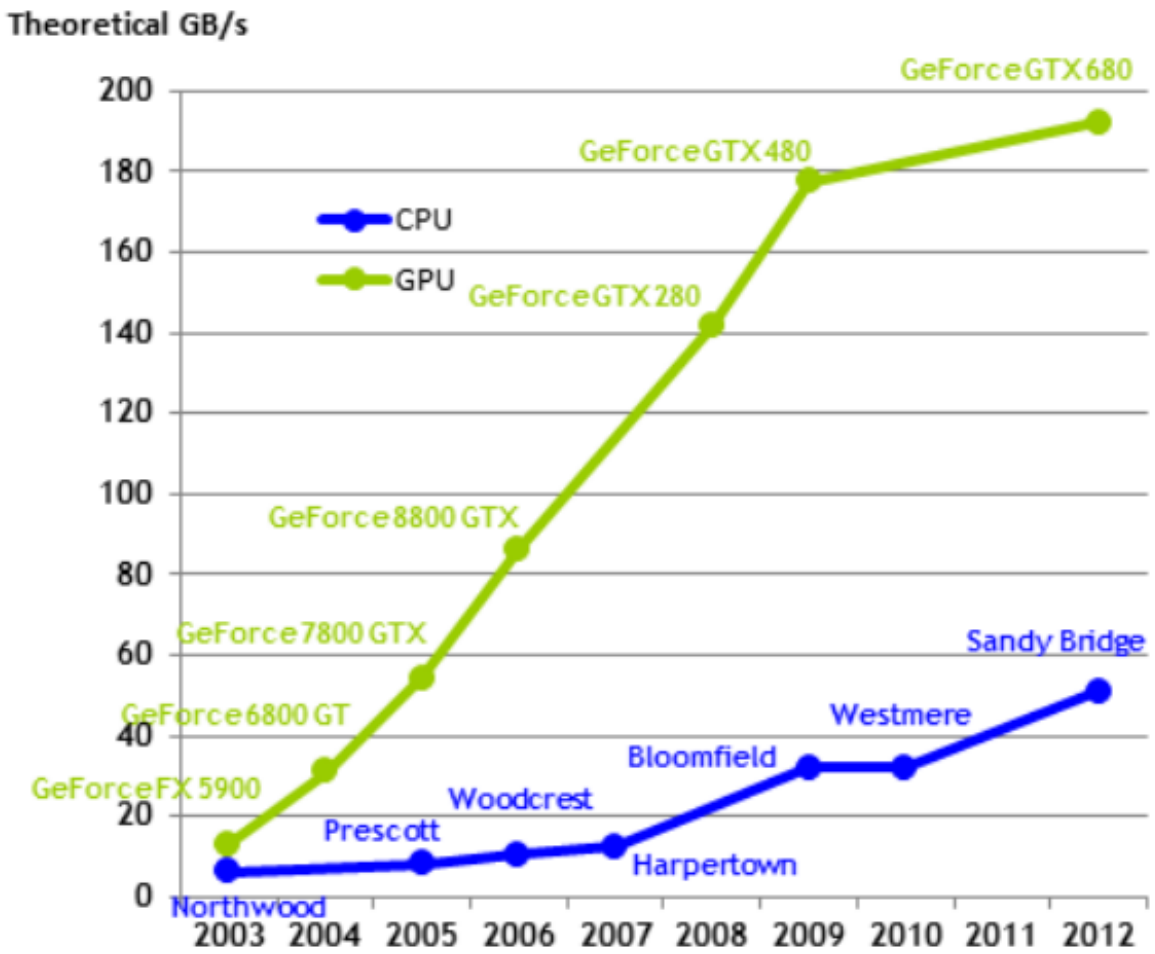
Intel Xeon Gold 6140 (released in 2017)

- 2 CPUs per motherboard
- 18 cores per CPU
- 2 AVX-512 (512 bits wide SIMD) FMA units per core
- clock speed : 2.3 GHz

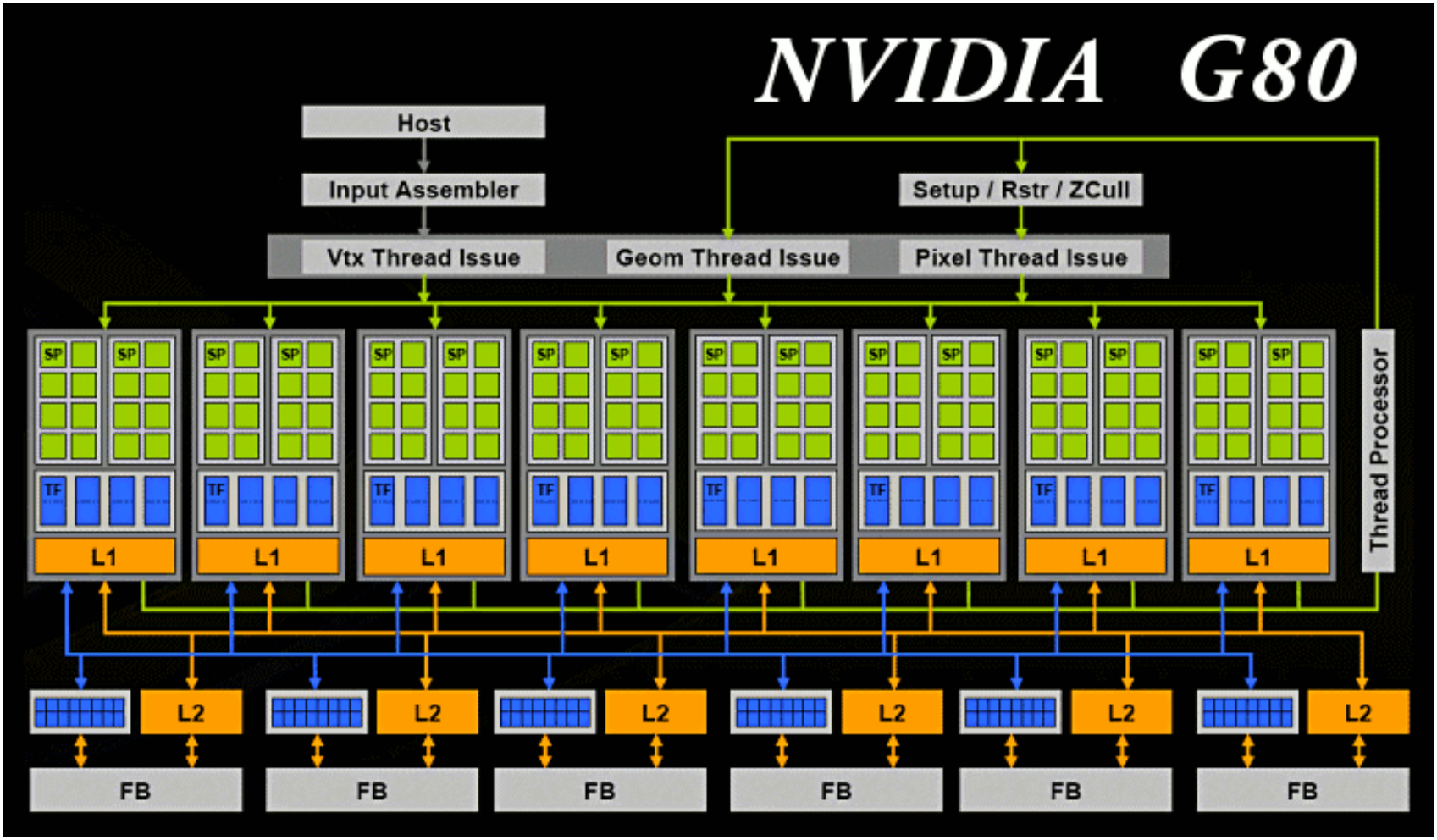
Theoretical double precision peak performance :

- number of double precision numbers per SIMD register : 8 (512/64)
- number of double precision operations per core per cycle : 32 ($8 \times 2 \times 2$)
- double precision gflops : 2.6 10^{12} ($32 \times 18 \times 2 \times 2.3\ 10^{9}$)

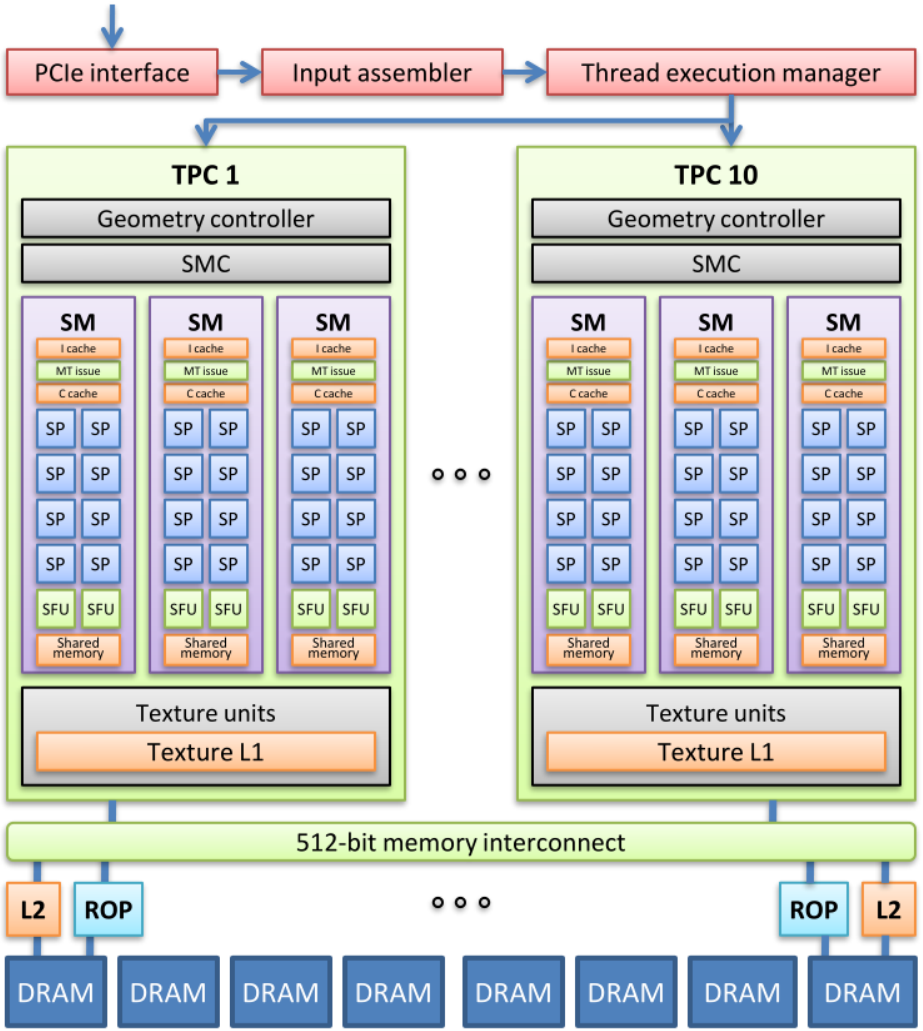
Memory bandwidth evolution



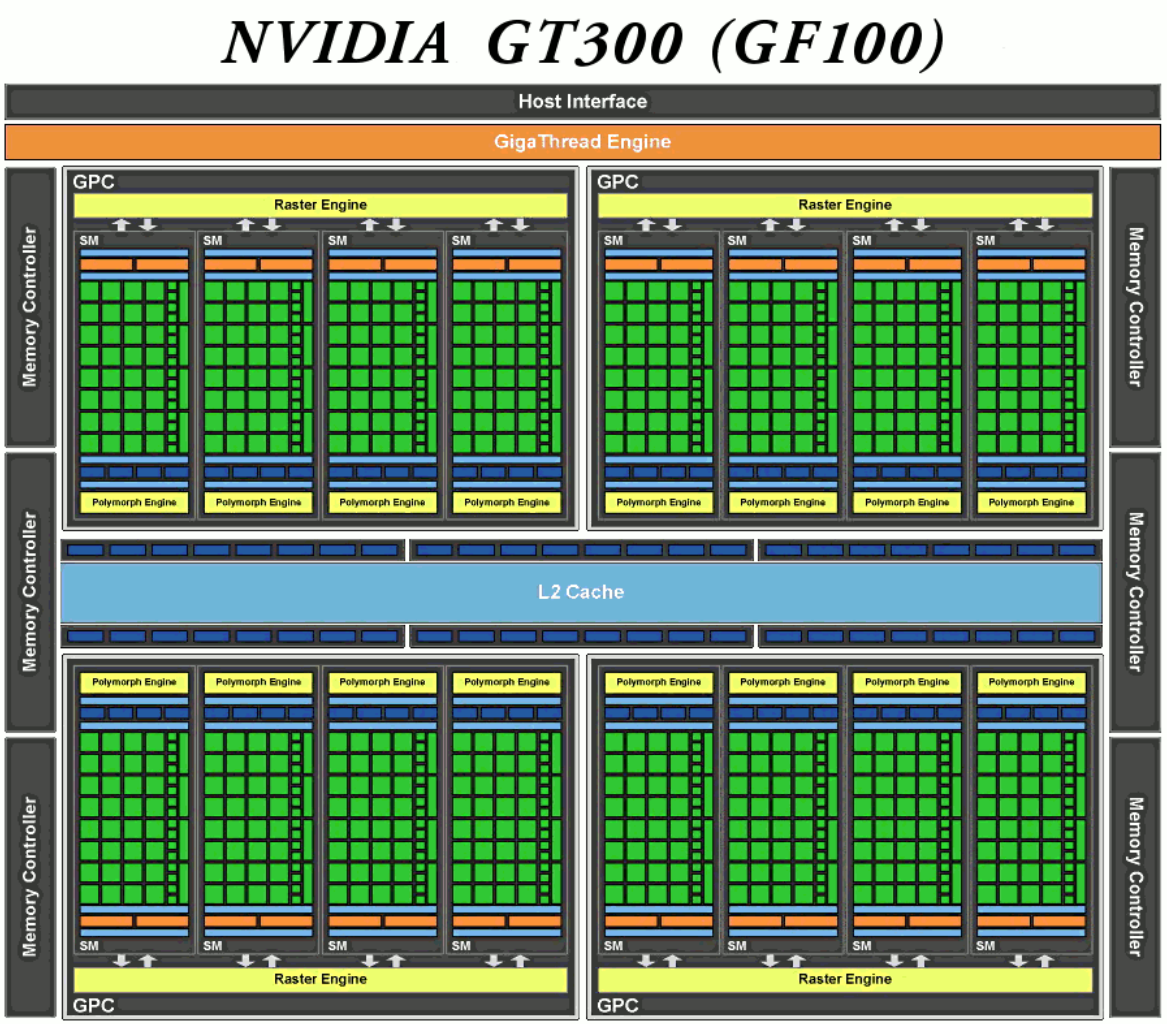
Nvidia G80 diagram



Nvidia GT200 diagram



Nvidia GF100 diagram



SM (Stream Multiprocessor) diagram

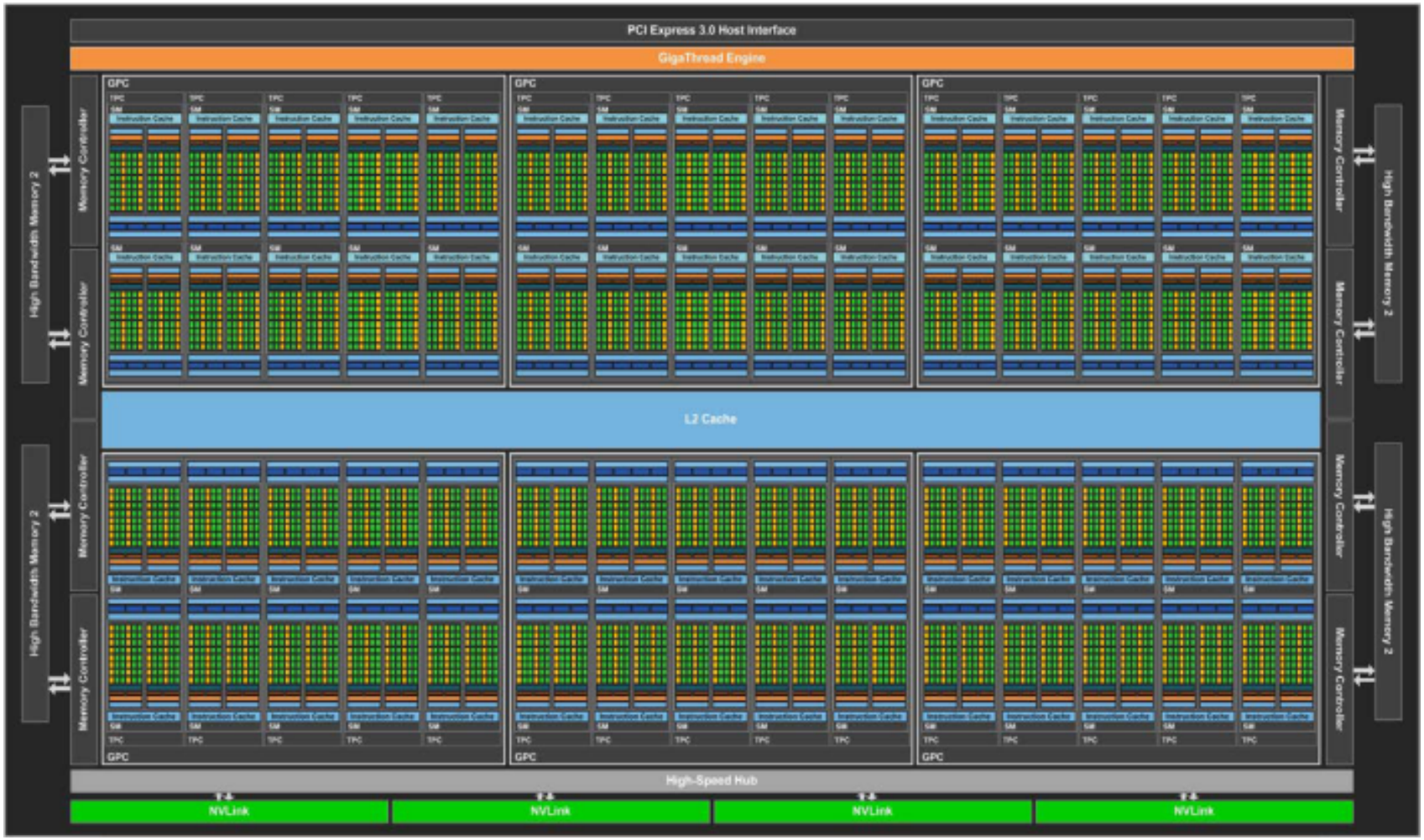
- 32 single precision CUDA cores
- or 16 double precision units
- 4 special function units (SFU)
- 16 load/store units (LD/ST)

Nvidia GK110 diagram

Kepler SMX Stream Multiprocessor diagram

- 192 single precision CUDA cores
- 64 double precision units
- 32 special function units (SFU)
- 32 load/store units (LD/ST)

Nvidia GP100 diagram



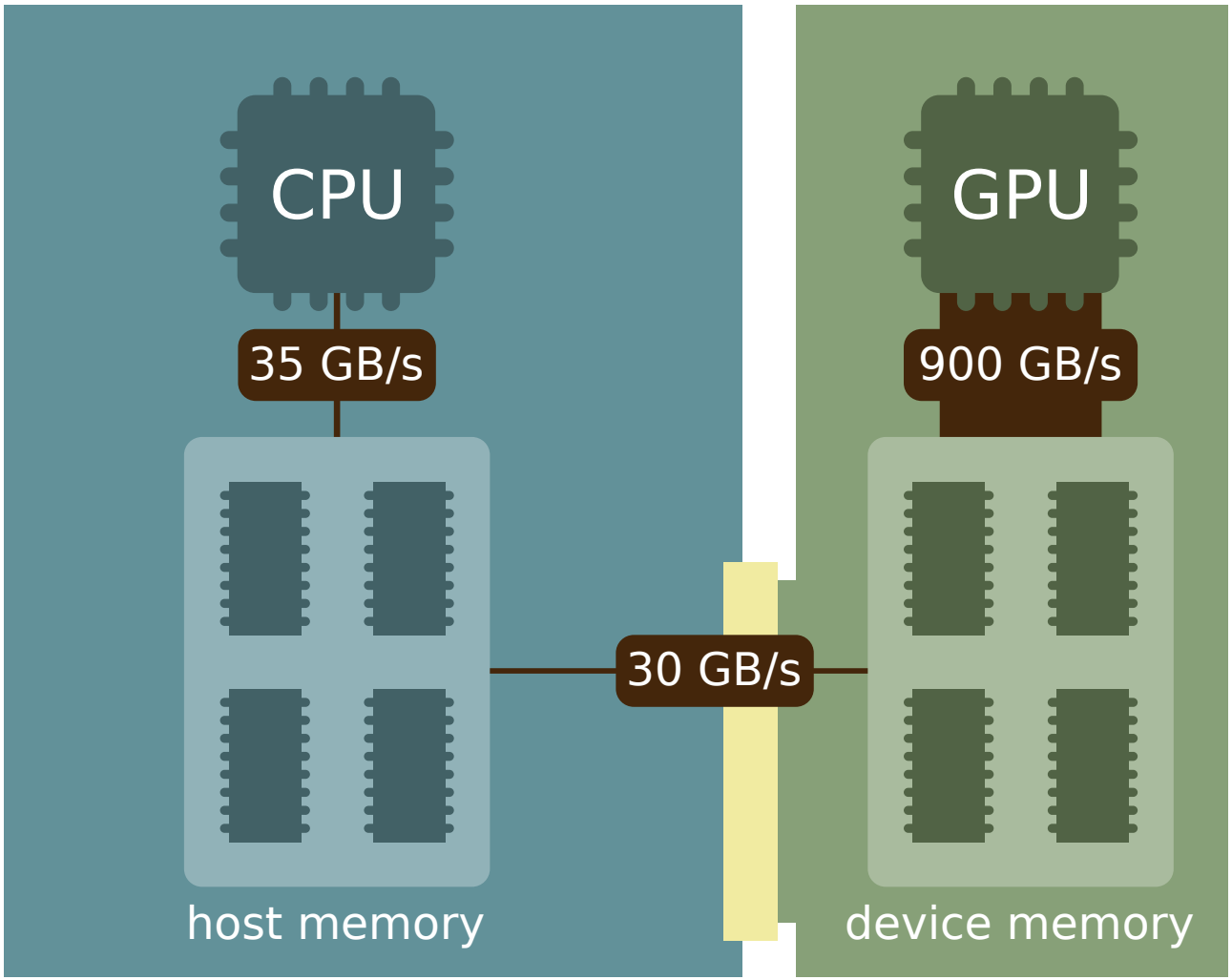
Pascal Stream Multiprocessor diagram

- 64 single precision CUDA cores
- 32 double precision units
- 16 special function units (SFU)
- 16 load/store units (LD/ST)

Operations per clock per multiprocessor

Compute capability Architecture	1.0, 1.1, 1.2 G80	1.3 Tesla	2.0 Fermi	2.1 Fermi	3.0 Kepler
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192
64-bit floating-point add, multiply, multiply-add	1	1	16	4	8
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm, base-2 exponential, sine, cosine	2	2	4	8	32
32-bit integer add	10	10	32	48	192
32-bit integer compare	10	10	32	48	160
32-bit integer shift	8	8	16	16	32
logical operations	8	8	32	48	160
32-bit integer multiply, multiply-add, sum of absolute difference	multiple instructions	multiple instructions	16	16	32

Typical CPU and GPU bandwidths



GPUs are not alone

- DSP
- FPGA (Field Programmable Gate Array)
- Intel Xeon Phi

Parallel programming languages

- (more or less) automatic parallelization
  - OpenMP
  - OpenHMPP
  - OpenAcc
- explicit parallelism
  - MPI (Message Passing Interface)
  - CUDA (Compute Unified Device Architecture)
  - OpenCL (Open Computing Language)
- implicit parallelism
  - SISAL
  - Parallel Haskell
  - Mitrion-C (for FPGAs)

Libraries

- libraries to ease writing parallel code (higher level than CUDA and openCL)
  - thrust
  - CUDPP
  - ArrayFire
- libraries of general purpose functions running on GPU
  - cuBLAS
  - cuFFT
  - cuLA
  - MAGMA

CUDA Example 1 : hello world !

main.cu

```
// This is the REAL "hello world" for CUDA!
// It takes the string "Hello ", prints it, then passes it >
     >to CUDA with an array
// of offsets. Then the offsets are added in parallel to >
     >produce the string "World!"
// By Ingemar Ragnemalm 2010

#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__ void hello(char *a, int *b)
{
   a[threadIdx.x] += b[threadIdx.x];
}

int main()
{
   char a[N] = "Hello \0\0\0\0\0\0\0\0\0\0\0\0";
   int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0,>
        > 0, 0};

   char *ad;
   int *bd;
   const int csize = N*sizeof(char);
   const int isize = N*sizeof(int);

   printf("%s", a);

   cudaMalloc( (void**)&ad, csize );
   cudaMalloc( (void**)&bd, isize );
   cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
   cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

   dim3 dimBlock( blocksize, 1 );
   dim3 dimGrid( 1, 1 );
   hello<<<dimGrid, dimBlock>>>(ad, bd);
   cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
   cudaFree( bd );
   cudaFree( ad );

   printf("%s\n", a);
   return EXIT_SUCCESS;
}
```

CUDA Example 2 : adding 2 vectors

main.cu (part 1/2)

```
__global__ void VecAddKernel(float* dA, float* dB, float* dC>
    >)
{
   int i = threadIdx.x;
   dC[i] = dA[i] + dB[i];
}

void VecAdd( const float* pA, const float* pB, float *pC, >
    >int vectorSize)
{
   size_t vectorMemSize = vectorSize * sizeof(float);

   // Allocate vectors in device memory
   float* dA;
   cudaMalloc( &dA, vectorMemSize );
   float* dB;
   cudaMalloc( &dB, vectorMemSize );
   float* dC;
   cudaMalloc( &dC, vectorMemSize );

   // copy vectors from host memory to device memory
   cudaMemcpy( dA, pA, vectorMemSize, cudaMemcpyHostToDevice>
       >);
   cudaMemcpy( dB, pB, vectorMemSize, cudaMemcpyHostToDevice>
       >);

   dim3 numBlocks(1,1,1);
   dim3 numThreadsPerBlock(vectorSize,1,1);
   // Kernel invocation
   VecAddKernel<<<numBlocks, numThreadsPerBlock>>>(dA, dB, >
       >dC);

   // copy the resulting vector from device memory to host >
       >memory
   cudaMemcpy( pC, dC, vectorMemSize, cudaMemcpyDeviceToHost>
       >);

   // free device memory
   cudaFree( dA );
   cudaFree( dB );
   cudaFree( dC );
}
```

CUDA Example 2: adding 2 vectors

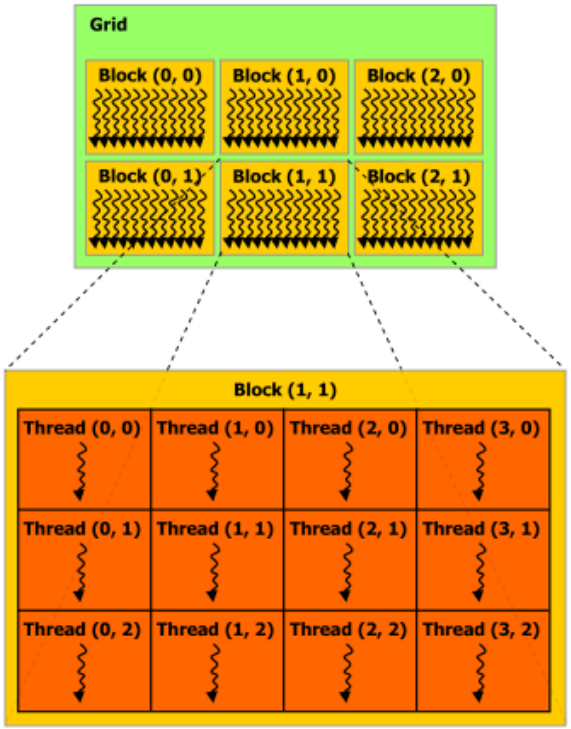
main.cu (part 2/2)

```
int main()
{
    int vectorSize = 100;
    size_t vectorMemSize = vectorSize * sizeof(float);

    // Allocate input vectors pA, pB and pC in host memory
    float * pA = (float*)malloc(vectorMemSize);
    float * pB = (float*)malloc(vectorMemSize);
    float * pC = (float*)malloc(vectorMemSize);

    // initialize input vectors
    ...

    VecAdd(pA, pB, pC, vectorSize);

    // use the resulting vector pC
    ...

    // and free the host memory
    free(pA);
    free(pB);
    free(pC);
}
```

Code explanation

- kernel : function that is executed on the gpu in parallel by N different CUDA threads
- a kernel is defined using the `__global__` declaration specifier
- `<<<...>>>` execution configuration syntax : specifies the number of CUDA threads that should be used (ie the number of calls to the kernel)
- Each thread that executes the kernel is given a unique thread identifier that is accessible within the kernel through the built-in threadIdx variable.

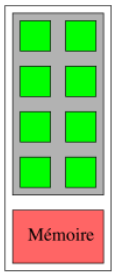
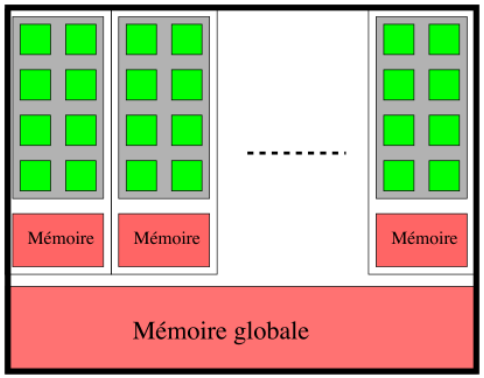
Thread Hierarchy



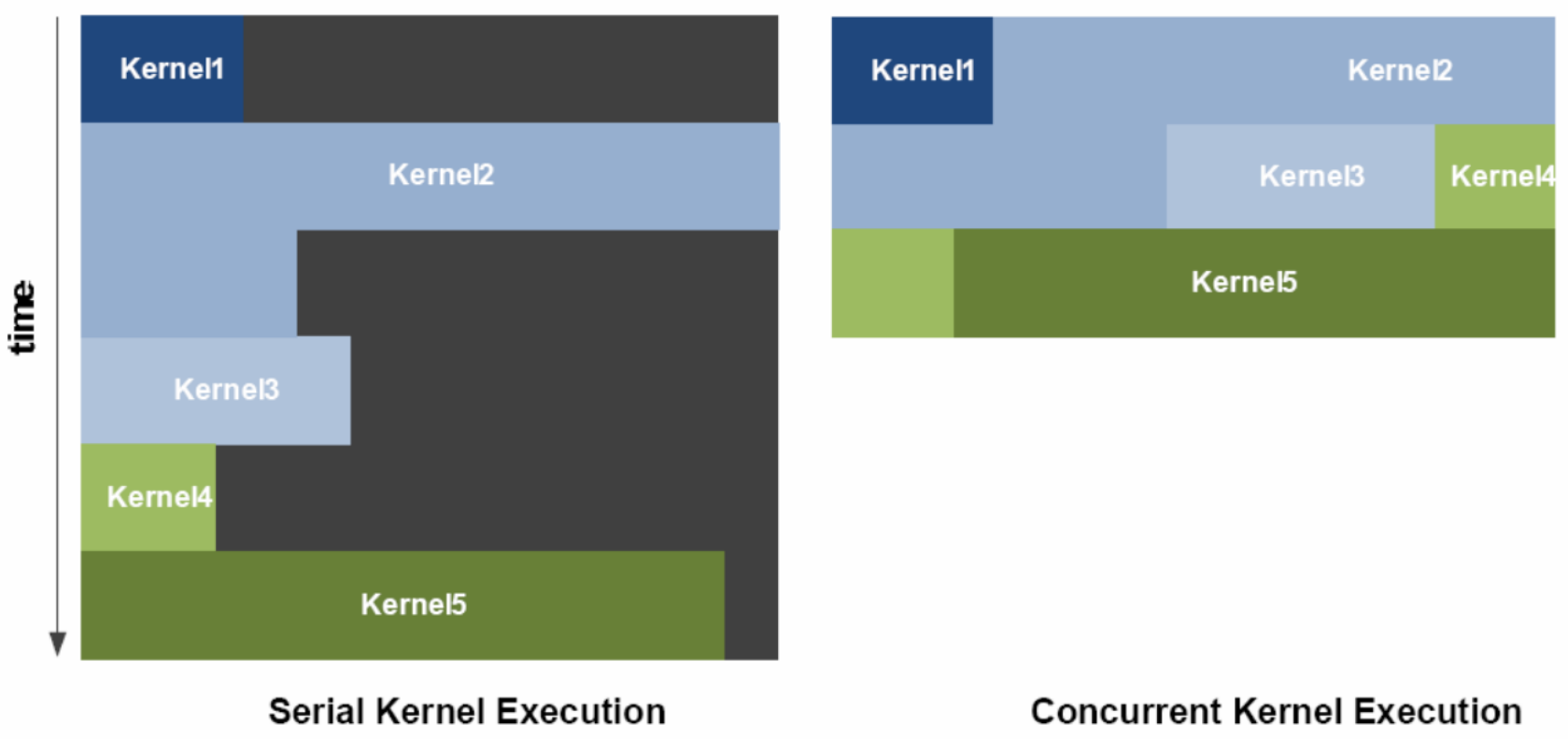
- threads are grouped into 1, 2 or 3-dimensional blocks
- blocks are grouped into a 1, 2 or 3-dimensional grid
- In a grid, all blocks have the same size
- the total number of threads is equal to the number of threads per block times the number of blocks
- all threads of a block execute on the same stream multiprocessor
- multiple thread blocks can execute concurrently on one multiprocessor
- A multiprocessor is designed to execute hundreds of threads concurrently
- Thread identification is provided to the kernel via the following built-in read-only variables :

```
dim3 threadIdx; // index of the thread inside its block
dim3 blockIdx; // index of the block inside the grid
dim3 blockDim; // size of the blocks (number of threads in a >
     >block, in each dimension)
dim3 gridDim; // size of the grid (number of blocks in the grid,>
     > in each dimension)
```

How kernels are executed on GPU

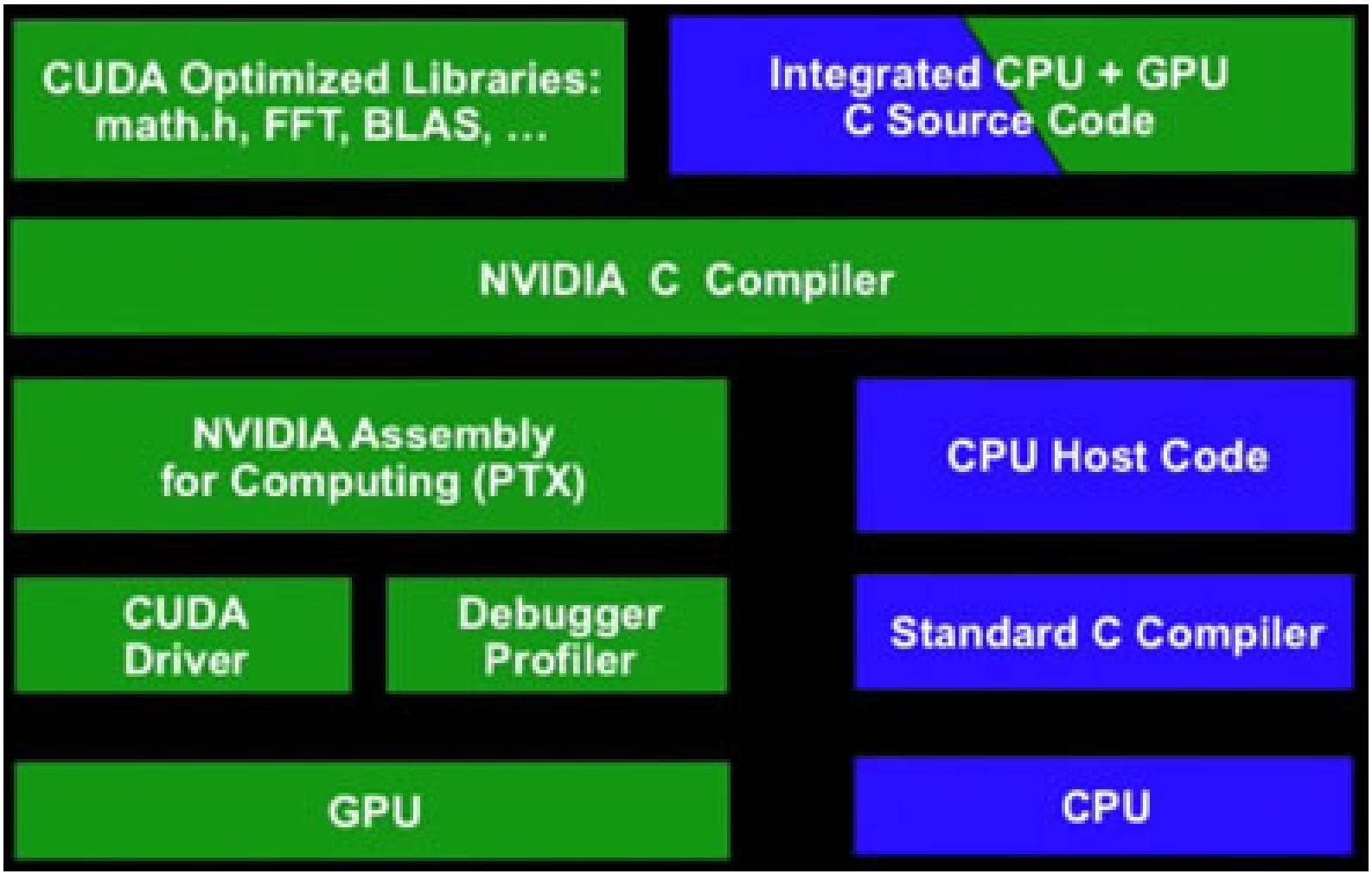
	Matériel	Logiciel	Exécution	
	un Stream Processor (SP)	**un thread**	séquentielle	**(a)**
	un coeur SIMT	**un bloc de threads** (plusieurs warps)	parallèle (SIMT)	**(b)**
	une carte GPU (device)	**une grille de threads** (kernel)	parallèle (MIMD) mémoire centralisée	**(c)**

Kernel execution on stream multiprocessors

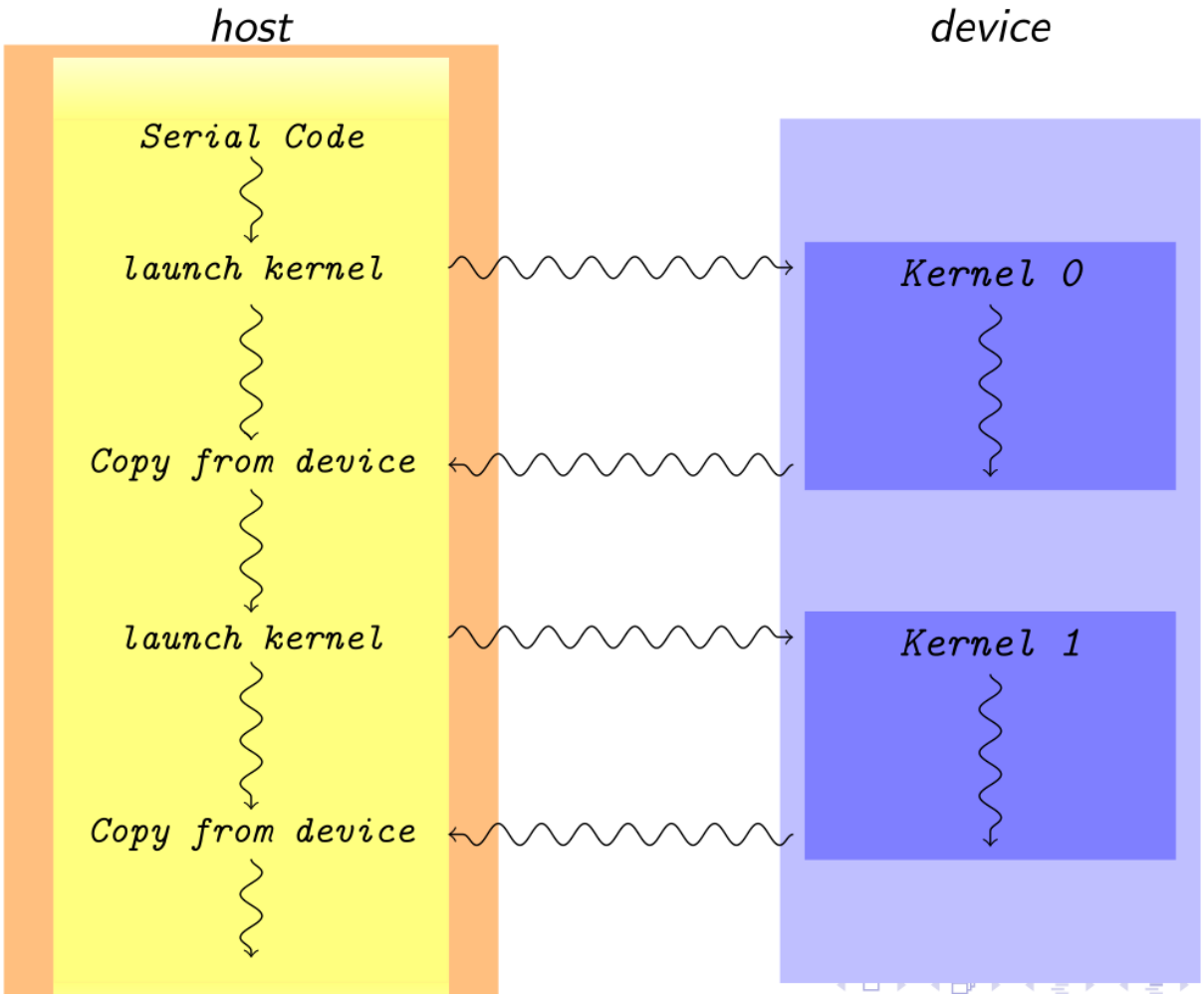


NVidia Fermi allows multiple kernels to be executed simultaneously.

Code compilation



Host and Device synchronization



CUDA Example 3 : adding 2 small square matrices

main.cu

```
__global__ void SmallMatAddKernel(float* pA, float* pB, >
      >float* pC, int matrixSize)
{
   int i = threadIdx.y * matrixSize + threadIdx.x;
   pC[i] = pA[i] + pB[i];
}

void SmallMatAdd( const float* pA, const float* pB, float *>
      >pC, int matrixSize)
{
   size_t matrixMemSize = matrixSize *matrixSize * sizeof(>
         >float);

   // Allocate matrices in device memory
   float* dA;
   cudaMalloc( &dA, matrixMemSize );
   float* dB;
   cudaMalloc( &dB, matrixMemSize );
   float* dC;
   cudaMalloc( &dC, matrixMemSize );

   // copy matrices from host memory to device memory
   cudaMemcpy( dA, pA, matrixMemSize, cudaMemcpyHostToDevice>
         >);
   cudaMemcpy( dB, pB, matrixMemSize, cudaMemcpyHostToDevice>
         >);

   // Kernel invocation with one block of N * N * 1 threads
   dim3 numBlocks(1,1,1);
   dim3 numThreadsPerBlock(matrixSize,matrixSize,1);
   // Kernel invocation
   SmallMatAddKernel<<<numBlocks, numThreadsPerBlock>>>(dA, >
         >dB, dC, matrixSize);

   // copy the resulting matrix from device memory to host >
         >memory
   cudaMemcpy( pC, dC, matrixMemSize, cudaMemcpyDeviceToHost>
         >);

   // free device memory
   cudaFree( dA );
   cudaFree( dB );
   cudaFree( dC );
}
```

CUDA Example 4 : adding 2 big square matrices (with constrained size)

main.cu

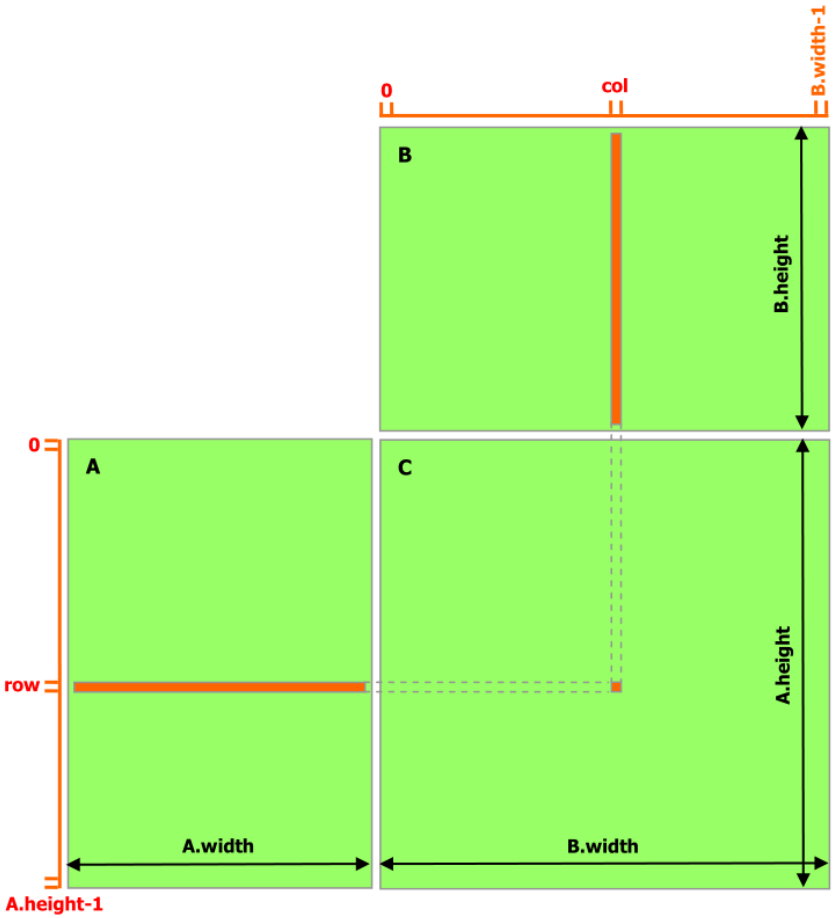
```
__global__ void BigMatAdd1Kernel(float* dA, float* dB, float>
    >* dC, int matrixSize)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   int i = y * matrixSize + x;
   dC[i] = dA[i] + dB[i];
}

void BigMatAdd1( const float* pA, const float* pB, float *pC>
    >, int matrixSize)
{
   size_t matrixMemSize = matrixSize *matrixSize * sizeof(>
       >float);

   // Allocate matrices in device memory
   float* dA;
   cudaMalloc( &dA, matrixMemSize );
   float* dB;
   cudaMalloc( &dB, matrixMemSize );
   float* dC;
   cudaMalloc( &dC, matrixMemSize );

   // copy matrices from host memory to device memory
   cudaMemcpy( dA, pA, matrixMemSize, cudaMemcpyHostToDevice>
       >);
   cudaMemcpy( dB, pB, matrixMemSize, cudaMemcpyHostToDevice>
       >);

   // Kernel invocation
   dim3 numThreadsPerBlock(16,16,1);
   dim3 numBlocks(matrixSize / numThreadsPerBlock.x, >
       >matrixSize / numThreadsPerBlock.y);
   // Kernel invocation
   BigMatAdd1Kernel<<<numBlocks, numThreadsPerBlock>>>(dA, >
       >dB, dC, matrixSize);

   // copy the resulting matrix from device memory to host >
       >memory
   cudaMemcpy( pC, dC, matrixMemSize, cudaMemcpyDeviceToHost>
       >);

   // free device memory
   cudaFree( dA );
   cudaFree( dB );
   cudaFree( dC );
}
```

CUDA Example 5 : adding 2 big matrices (without constrained size)

main.cu

```
__global__ void BigMatAdd2Kernel(float* dA, float* dB, float>
     >* dC, int matrixSize)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < matrixSize && y < matrixSize)
   {
      int i = y * matrixSize + x;
      dC[i] = dA[i] + dB[i];
   }
}

int getNumBlocks(int numElements, int blockSize ) { return (>
     >numElements - 1)/blockSize + 1; }

void BigMatAdd2( const float* pA, const float* pB, float *pC>
     >, int matrixSize)
{
   size_t matrixMemSize = matrixSize *matrixSize * sizeof(>
        >float);

   // Allocate matrices in device memory
   float* dA;
   cudaMalloc( &dA, matrixMemSize );
   float* dB;
   cudaMalloc( &dB, matrixMemSize );
   float* dC;
   cudaMalloc( &dC, matrixMemSize );

   // copy matrices from host memory to device memory
   cudaMemcpy( dA, pA, matrixMemSize, cudaMemcpyHostToDevice>
        >);
   cudaMemcpy( dB, pB, matrixMemSize, cudaMemcpyHostToDevice>
        >);

   // Kernel invocation
   dim3 numThreadsPerBlock(16,16,1);
   dim3 numBlocks(getNumBlocks(matrixSize, >
        >numThreadsPerBlock.x), getNumBlocks(matrixSize, >
        >numThreadsPerBlock.y), 1);
   // Kernel invocation
   BigMatAdd2Kernel<<<numBlocks, numThreadsPerBlock>>>(dA, >
        >dB, dC, matrixSize);

   // copy the resulting matrix from device memory to host >
        >memory
   cudaMemcpy( pC, dC, matrixMemSize, cudaMemcpyDeviceToHost>
        >);

   // free device memory
   cudaFree( dA );
   cudaFree( dB );
   cudaFree( dC );
}
```

CUDA Example 6 : multiply 2 big matrices (slow version)



$$C = AB$$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Complexity : $O(N^3)$

CUDA Example 6 : multiply 2 big matrices (slow version)

main.cu (1/2)

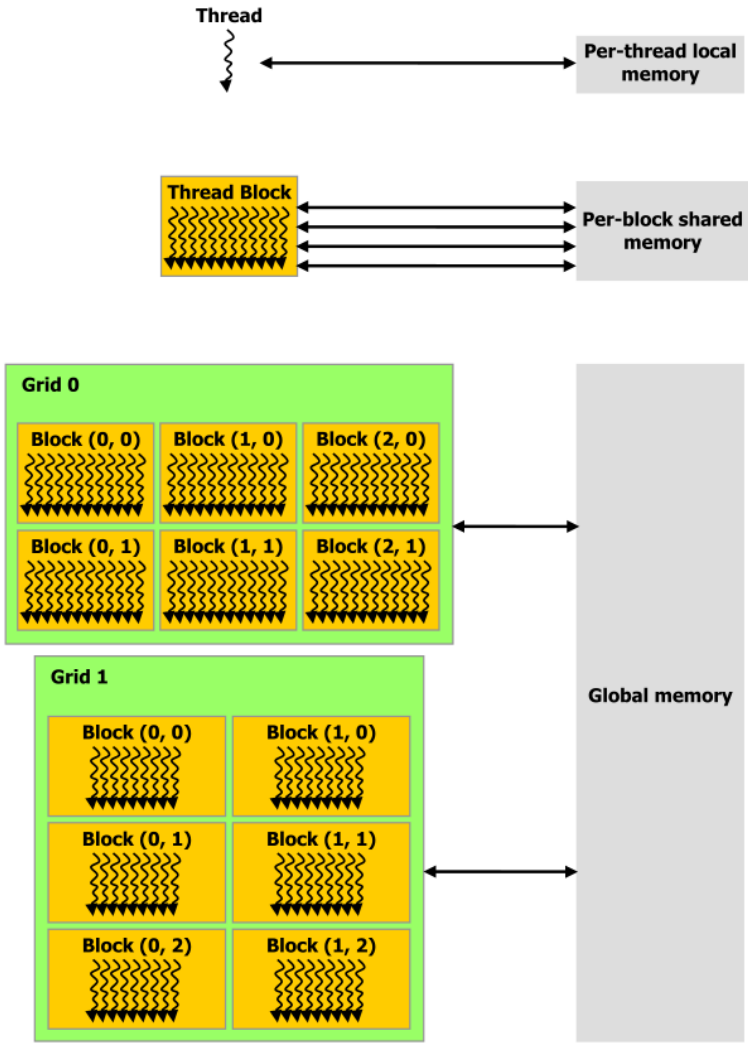
```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
   int width;
   int height;
   float* elements;
} Matrix;

// Matrix multiplication kernel called by MatMul()
__global__ void SlowMatMulKernel(Matrix A, Matrix B, >
      >Matrix C)
{
   // Each thread computes one element of C
   // by accumulating results into Cvalue
   float Cvalue = 0;
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   for (int e = 0;e < A.width;++e)
      Cvalue += A.elements[row * A.width + e] * B.>
            >elements[e * B.width + col];
   C.elements[row * C.width + col] = Cvalue;
}
```

CUDA Example 6 : multiply 2 big matrices (slow version)

main.cu (2/2)

```
// Thread block size
#define BLOCK_SIZE 16

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of >
     >BLOCK_SIZE
void SlowMatMul(const Matrix A, const Matrix B, Matrix C>
     >)
{
   // Load A and B to device memory
   Matrix dA;
   dA.width = A.width;dA.height = A.height;
   size_t size = A.width * A.height * sizeof(float);
   cudaMalloc(&dA.elements, size);
   cudaMemcpy(dA.elements, A.elements, size, >
        >cudaMemcpyHostToDevice);
   Matrix dB;
   dB.width = B.width;dB.height = B.height;
   size = B.width * B.height * sizeof(float);
   cudaMalloc(&dB.elements, size);
   cudaMemcpy(dB.elements, B.elements, size, >
        >cudaMemcpyHostToDevice);

   // Allocate C in device memory
   Matrix dC;
   dC.width = C.width;dC.height = C.height;
   size = C.width * C.height * sizeof(float);
   cudaMalloc(&dC.elements, size);

   // Invoke kernel
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(B.width / dimBlock.x, A.height / >
        >dimBlock.y);
   SlowMatMulKernel<<<dimGrid, dimBlock>>>(dA, dB, dC);

   // Read C from device memory
   cudaMemcpy(C.elements, dC.elements, size, >
        >cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(dA.elements);
   cudaFree(dB.elements);
   cudaFree(dC.elements);
}
```

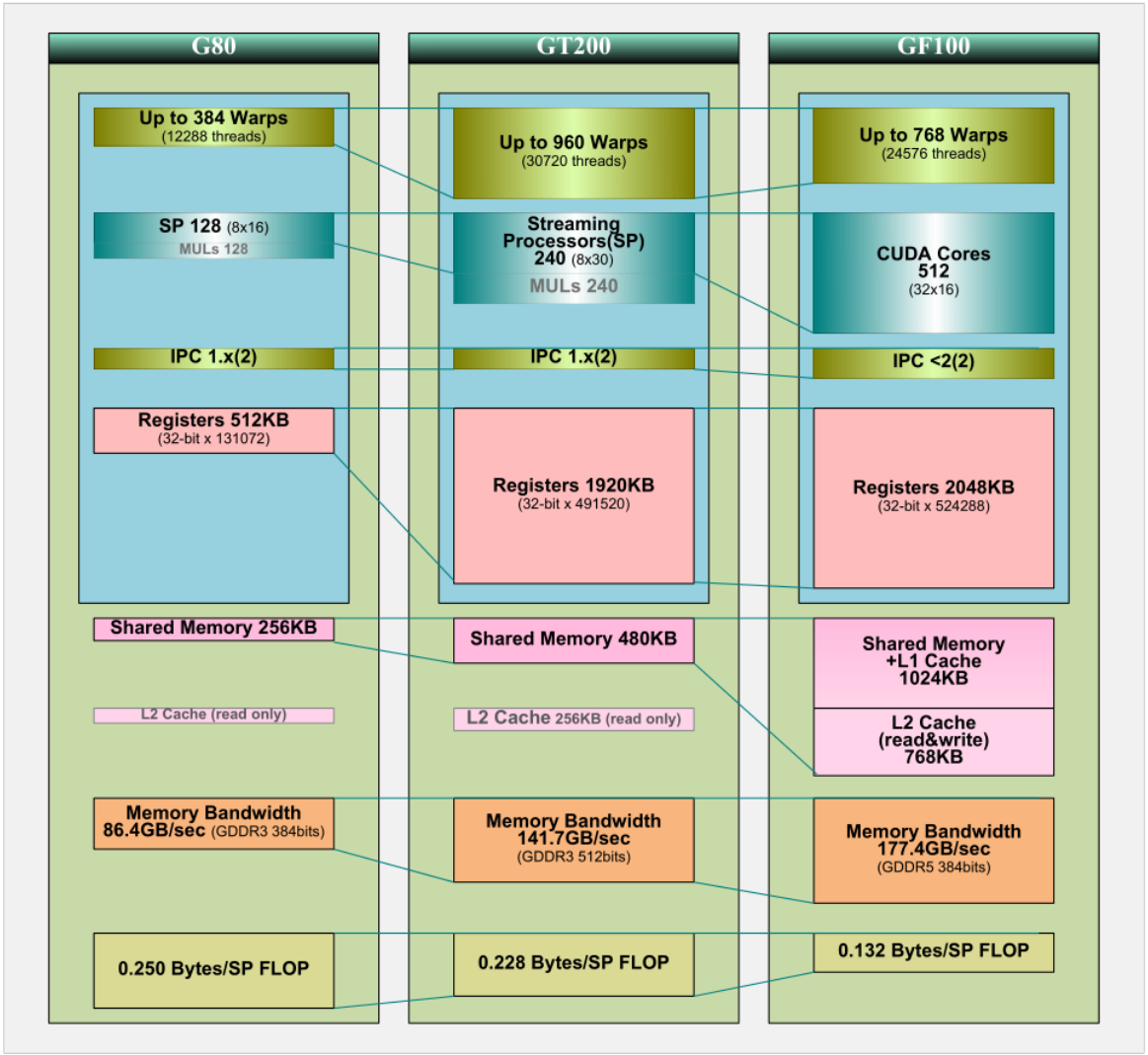
Memory hierarchy



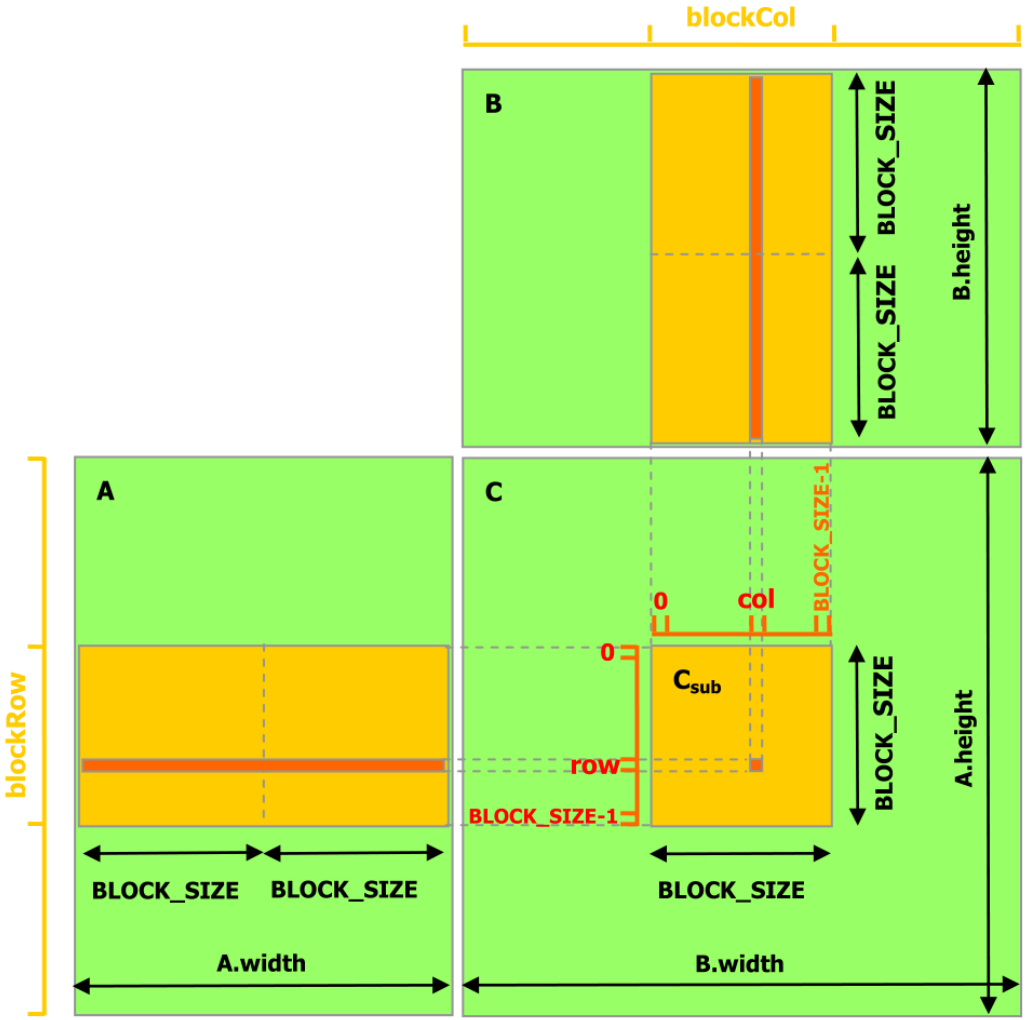
How to improve efficiency of matrix multiplication ?

- global memory latency : about 500 cycles !!!
- ”Shared memory is expected to be much faster than global memory”
- ”Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited”

CUDA architecture comparison



CUDA Example 7 : multiply 2 big matrices (efficient version)



CUDA Example 7 : multiply 2 big matrices (efficient version)

main.cu (1/3)

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
   int width;
   int height;
   int stride;
   float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int>
      > col)
{
   return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, >
      >float value)
{
   A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A >
      >that is
// located col sub-matrices to the right and row sub->
      >matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int >
      >col)
{
   Matrix Asub;
   Asub.width    = BLOCK_SIZE;
   Asub.height   = BLOCK_SIZE;
   Asub.stride   = A.stride;
   Asub.elements = &A.elements[A.stride * BLOCK_SIZE * >
         >row + BLOCK_SIZE * col];
   return Asub;
}
```

CUDA Example 7 : multiply 2 big matrices (efficient version)

main.cu (2/3)

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix >
    >C)
{
   // Each thread block computes one sub-matrix Csub of >
       >C
   Matrix Csub = GetSubMatrix(C, blockIdx.y, blockIdx.x)>
       >;
   // Each thread computes one element of Csub by >
       >accumulating results into Cvalue
   float Cvalue = 0;

   // Loop over all the sub-matrices of A and B that are>
       > required to compute Csub
   // Multiply each pair of sub-matrices together and >
       >accumulate the results
   for (int m = 0;m < (A.width / BLOCK_SIZE);++m)
   {
      // Get sub-matrix Asub of A
      Matrix Asub = GetSubMatrix(A, blockIdx.y, m);

      // Get sub-matrix Bsub of B
      Matrix Bsub = GetSubMatrix(B, m, blockIdx.x);

      // Shared memory used to store Asub and Bsub >
          >respectively
      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

      // Load Asub and Bsub from device memory to shared>
          > memory
      // Each thread loads one element of each sub->
          >matrix
      As[threadIdx.y][threadIdx.x] = GetElement(Asub, >
          >threadIdx.y, threadIdx.x);
      Bs[threadIdx.y][threadIdx.x] = GetElement(Bsub, >
          >threadIdx.y, threadIdx.x);

      // Synchronize to make sure the sub-matrices are >
          >loaded before starting the computation
      __syncthreads();

      // Multiply Asub and Bsub together
      for (int e = 0;e < BLOCK_SIZE;++e)
         Cvalue += As[threadIdx.y][e] * Bs[e][threadIdx.>
             >x];

      // Synchronize to make sure that the preceding >
          >computation is done before loading two new >
          >sub-matrices of A and B in the next >
          >iteration
      __syncthreads();
   }

   // Write Csub to device memory
   // Each thread writes one element
   SetElement(Csub, threadIdx.y, threadIdx.x, Cvalue);
}
```

CUDA Example 7 : multiply 2 big matrices (efficient version)

main.cu (3/3)

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of >
     >BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
   // Load A and B to device memory
   Matrix dA;
   dA.width = dA.stride = A.width;dA.height = A.height;
   size_t size = A.width * A.height * sizeof(float);
   cudaMalloc(&dA.elements, size);
   cudaMemcpy(dA.elements, A.elements, size, >
        >cudaMemcpyHostToDevice);
   Matrix dB;
   dB.width = dB.stride = B.width;dB.height = B.height;
   size = B.width * B.height * sizeof(float);
   cudaMalloc(&dB.elements, size);
   cudaMemcpy(dB.elements, B.elements, size, >
        >cudaMemcpyHostToDevice);

   // Allocate C in device memory
   Matrix dC;
   dC.width = dC.stride = C.width;dC.height = C.height;
   size = C.width * C.height * sizeof(float);
   cudaMalloc(&dC.elements, size);

   // Invoke kernel
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(B.width / dimBlock.x, A.height / >
        >dimBlock.y);
   MatMulKernel<<<dimGrid, dimBlock>>>(dA, dB, dC);

   // Read C from device memory
   cudaMemcpy(C.elements, dC.elements, size, >
        >cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(dA.elements);
   cudaFree(dB.elements);
   cudaFree(dC.elements);
}
```

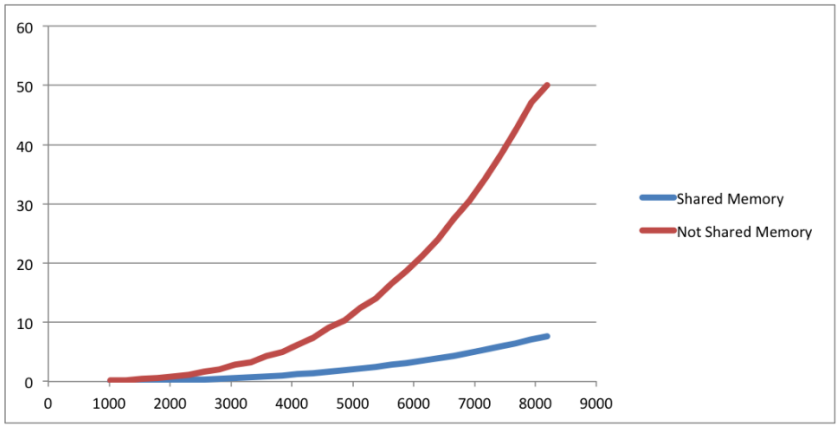
CUDA Example 7 : multiply 2 big matrices (efficient version)

Benefits :

- each element of submatrix ASub is re-used `BLOCK_SIZE` times : in other words, the number of global memory accesses for matrix A is divided by `BLOCK_SIZE`
- the same goes for matrix B

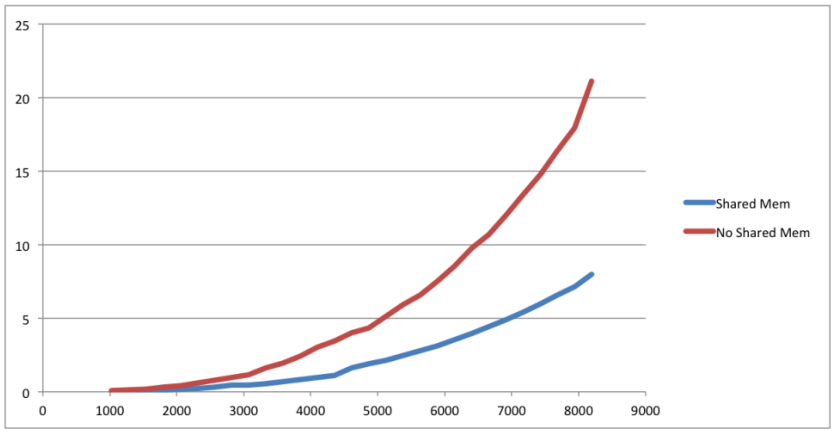
Performance comparison

Tesla C1060



Running time vs. matrix size.

Fermi M2070



Running time vs. matrix size.

- 8000^3 multiplications
- 8000^3 additions
- in 8 seconds

Performance analysis for Fermi M2070 :

memory usage : $8000^2 \times 4(\text{sizeof(float)}) \times 3(\text{num matrices}) = 768$ MB

measured performance : $\frac{8000^3+8000^3}{8} = 128.10^9$ operations per second

theoretical peak performance (SP) : $\approx 1000.10^9$ operations per second

Generated code

main.ptx (partial view)

```
bar.sync        0;
 .loc    3       70      0
 ld.shared.f32   %f4, [%rd8+0];
 ld.shared.f32   %f5, [%rd6+0];
 mad.f32         %f6, %f4, %f5, %f1;
 ld.shared.f32   %f7, [%rd8+4];
 ld.shared.f32   %f8, [%rd6+64];
 mad.f32         %f9, %f7, %f8, %f6;
 ... (14 similar groups of 3 lines)
 ld.shared.f32   %f46, [%rd8+56];
 ld.shared.f32   %f47, [%rd6+896];
 mad.f32         %f48, %f46, %f47, %f45;
 ld.shared.f32   %f49, [%rd8+60];
 ld.shared.f32   %f50, [%rd6+960];
 mad.f32         %f1, %f49, %f50, %f48;
 .loc    3       73      0
 bar.sync        0;
```

CUDA Example 8 : reduction

- NVIDIA Presentation : Optimizing Parallel Reduction in CUDA
- Author : Mark Harris

Example of applications

- Physics and chemistry
  - Atmospheric science. Cloud simulation.
  - Ocean and space science.
  - Computational electromagnetics and electrodynamics.
  - Computational electrostatics. ion placement.
  - Computational fluid dynamics. Euler solvers for nonlinear PDEs (Navier-Stokes equations).
  - Quantum chemistry. Electron repulsion integrals.
  - Astrophysics. N-body simulations.
  - Molecular dynamics. Non-bonded force calculations.
- Energy, gas, and oil exploration
  - 3D seismic analysis. FFT.
  - Geophysical visualization. Filtering and reducing raw seismic data.
- Life sciences
  - Protein folding (Stanford folding@home project).
  - Smith-Waterman sequence alignment. Dynamic programming for approximate alignment.
  - MUMmerGPU exact sequence alignment using suffix trees.
  - HMMer protein sequence analysis.
  - Combinatorial chemistry for drug design.
  - Hidden Markov models.
  - Protein docking.
  - Fluorescence microscopy simulation.
  - Neuron simulation.

Reported GPU application speedups (2009)

- 1080x: Monte Carlo simulation of photon migration
- 675x: stochastic differential equations
- 470x: k nearest neighbor search
- 420x: generalized harmonic analysis
- 340x: power system simulation
- 300x: cone beam computed tomography
- 270x: Particle swarm optimization
- 263x: FHD-spiral MRI reconstruction
- 250x: N-body simulation
- 172x: support vector machine training and classification
- 169x: signal and image reconstruction
- 130x: quantum chemistry two-electron integral evolution
- 120x: 3D particle Boltzmann solver
- 109x: sliding window object detection
- 100x: visualization of volumetric white matter connectivity
- 100x: prestack seismic data interaction
- 100x: folding@home
- 100x: pricing and risk management of exotic financial structures
- 100x: incompressible Navier-Stokes solver

If you want to know more...

- NVIDIA : NVidia Cuda C Programming Guide v4.2
- NVIDIA : Cuda C Best practices Guide v4.1
- Vasily Volkov : Better Performance at Lower Occupancy