Reorder Buffer (RoB)

- Register values and memory values are not written until an instruction commits

- RoB effectively renames the destination registers
  - every destination gets a new entry in the RoB
- On misprediction:
  - Speculated entries in RoB are cleared

- Exceptions:
  - Not recognized/taken until it is ready to commit
  - Precise exceptions require that 'later' entries in RoB are cleared

Flynn Taxonomy, 1966*

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

Flynn* Taxonomy, 1966

- In 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination using synchronization primitives

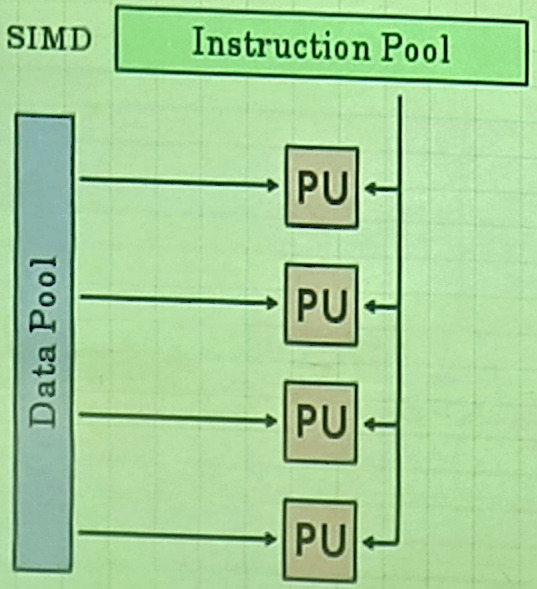
		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

Flynn* Taxonomy, 1966

- In 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination using synchronization primitives
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

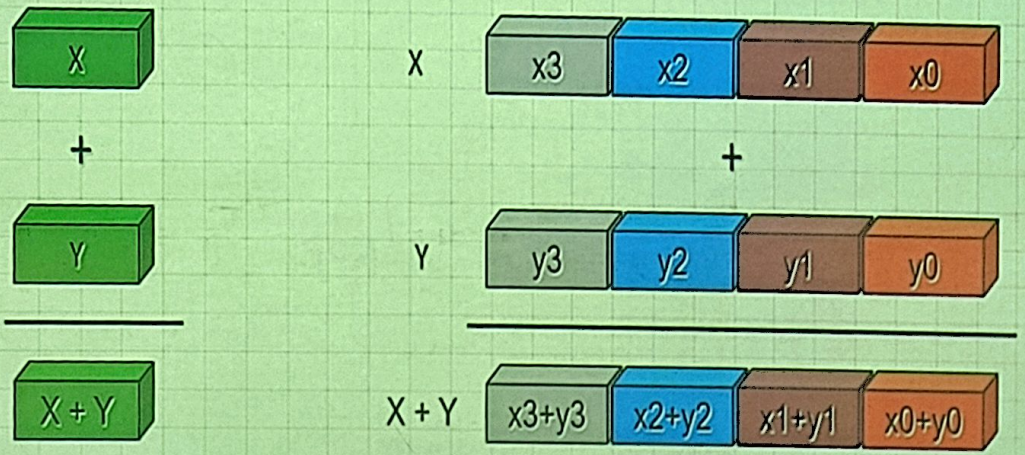
Single-Instruction/Multiple-Data Stream (SIMD or "sim-dee")



- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

SIMD: Single Instruction, Multiple Data

- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - With Intel SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

What does this mean to you?

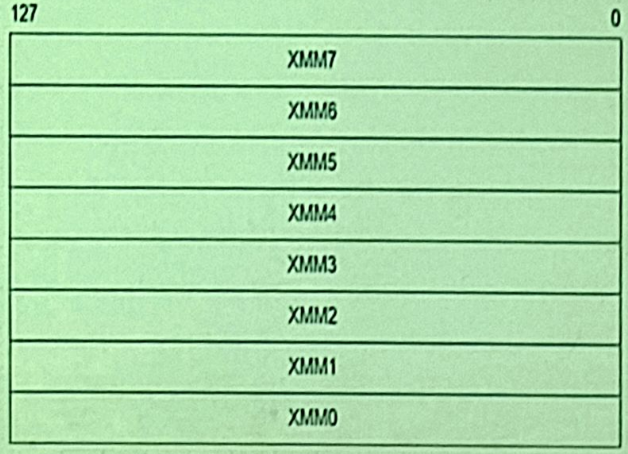
- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:

    $x = y + c * z$

    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Intel SIMD Extensions

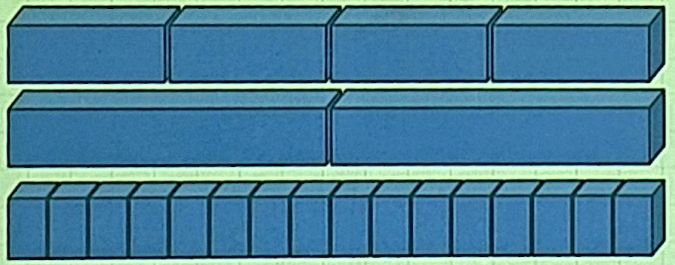
- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 8 128-bit registers [1999]



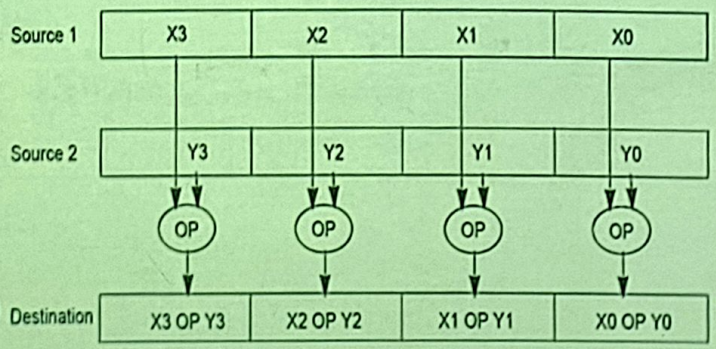
- AVX, new 256-bit registers [2011]
  - Space for expansion to 1024-bit registers

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



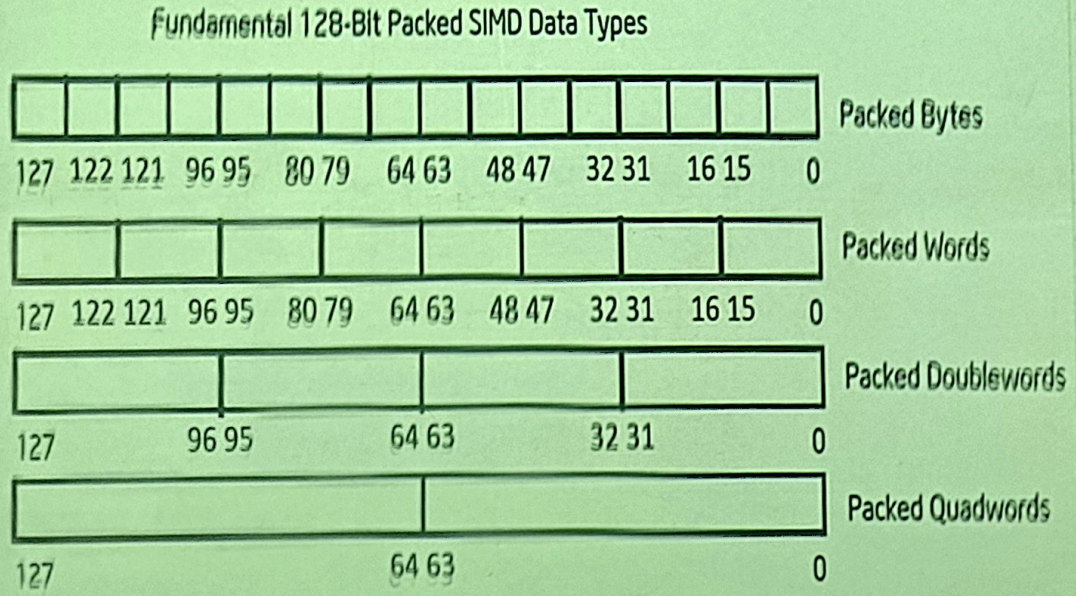
- Instructions perform add, multiply etc. on all the data in parallel



- Similar on GPUs, vector p erations)

Intel Architecture SSE2+ 128-Bit SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single-precision FP: Double word (32 bits)
  - Double-precision FP: Quad word (64 bits)



Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

```
for each f in array
{
    load f to floating-point register
    calculate the square root
    write the result from the
register to memory
}
```

```
for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    store the 4 results from the register to memory
}
```

SIMD style

Data-Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- How can reveal more data-level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Could be rewritten

```
for(i=1000; i>0; i=i-4) {
        x[i]   = x[i] + s;
        x[i-1] = x[i-1] + s;
        x[i-2] = x[i-2] + s;
        x[i-3] = x[i-3] + s;
  }
```

Building a Python-C extension

- Write the C function
  - PyObject
  - PyArg_ParseTuple()
  - PyLong_FromLong()

- First import calls this

Building a Python-C extension

- Write the C function
  - PyObject
  - PyArg_ParseTuple()
  - PyLong_FromLong()
- Write the init function
  - PyMethodDef
  - PyModuleDef
  - First import calls this

```c
#include <Python.h>

static PyObject *method_fputs(PyObject *self, PyObject *args) {
    char *str, *filename = NULL;
    int bytes_copied = -1;

    /* Parse arguments */
    if(!PyArg_ParseTuple(args, "ss", &str, &filename)) {
        return NULL;
    }

    FILE *fp = fopen(filename, "w");
    bytes_copied = fputs(str, fp);
    fclose(fp);

    return PyLong_FromLong(bytes_copied);
}

static PyMethodDef FputsMethods[] = {
    {"fputs", method_fputs, METH_VARARGS, "Python interface for fputs
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef fputsmodule = {
    PyModuleDef_HEAD_INIT,
    "fputs",
    "Python interface for the fputs C library function",
    -1,
    FputsMethods
};
```

thon-C extension

```
#include <Python.h>

static PyObject *method_fputs(PyObject *self, PyObject *args) {
    char *str, *filename = NULL;
    int bytes_copied = -1;

    /* Parse arguments */
    if(!PyArg_ParseTuple(args, "ss", &str, &filename)) {
        return NULL;
    }

    FILE *fp = fopen(filename, "w");
    bytes_copied = fputs(str, fp);
    fclose(fp);

    return PyLong_FromLong(bytes_copied);
}

static PyMethodDef FputsMethods[] = {
    {"fputs", method_fputs, METH_VARARGS,
```

```
    FILE *fp = fopen(filename, "w");
    bytes_copied = fputs(str, fp);
    fclose(fp);

    return PyLong_FromLong(bytes_copied);
}

static PyMethodDef FputsMethods[] = {
    {"fputs", method_fputs, METH_VARARGS, "Python interface for fputs
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef fputsmodule = {
    PyModuleDef_HEAD_INIT,
    "fputs",
    "Python interface for the fputs C library function",
    -1,
    FputsMethods
;
```

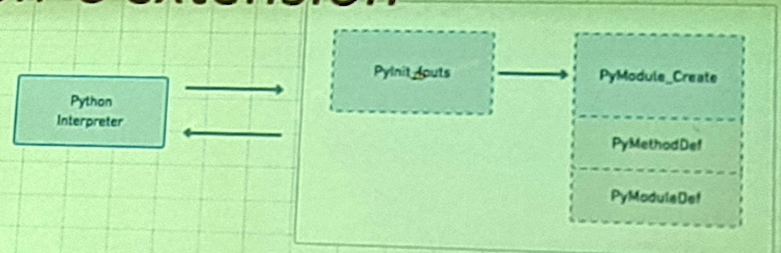
- Write the init function
  - PyMethodDef
  - PyModuleDef
- PyMODINIT_FUNC
  - First import calls this

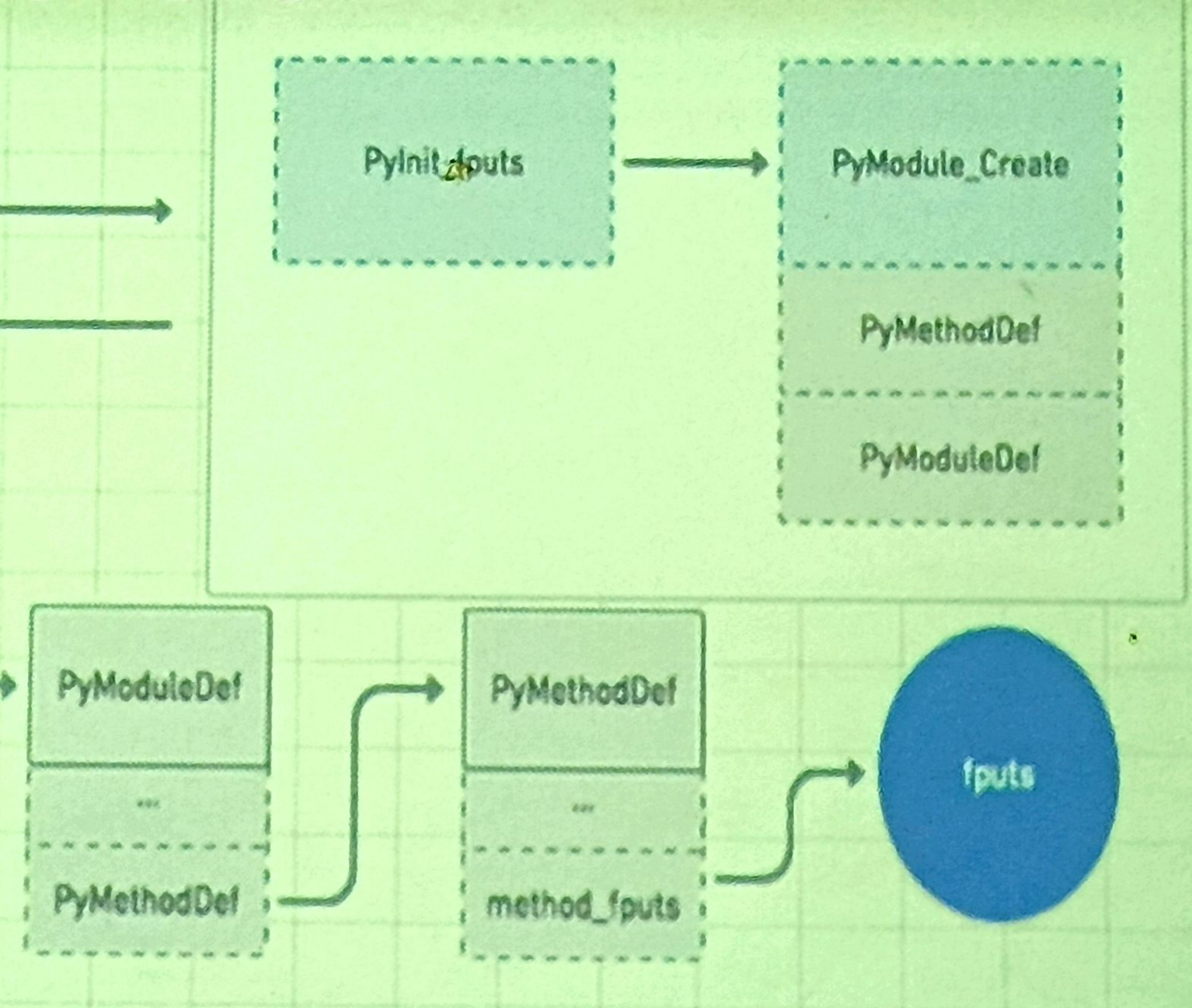
```
PyMODINIT_FUNC PyInit_fputs(void) {
    return PyModule_Create(&fputsmodule);
}
```

```
    FILE *fp = fopen(filename, "w");
    bytes_copied = fputs(str, fp);
    fclose(fp);

    return PyLong_FromLong(bytes_copied);
}

static PyMethodDef FputsMethods[] = {
    {"fputs", method_fputs, METH_VARARGS, "Python interface for fputs
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef fputsmodule = {
    PyModuleDef_HEAD_INIT,
    "fputs",
    "Python interface for the fputs C library function",
    -1,
    FputsMethods
};
```

Building a Python-C extension

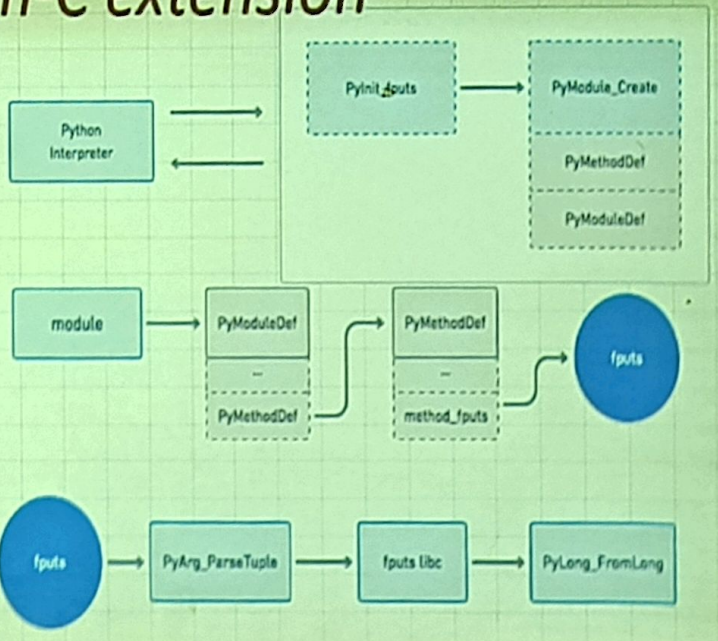
- This is how it appears




PyInit_fputs
PyModule_Create
PyMethodDef
PyModuleDef
PyModuleDef
...
PyMethodDef
PyMethodDef
...
method_fputs
fputs

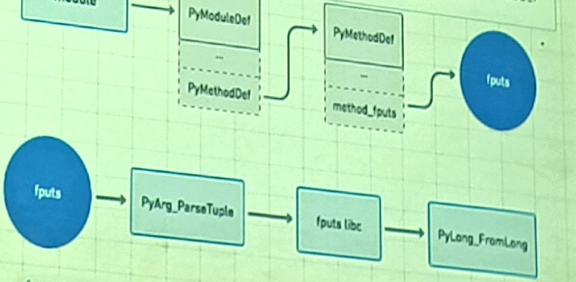
Building a Python-C extension

- This is how it appears
- Create setup.py
- Compile and Install
  - python3_setup.py_install



– python3_setup.py_install

Use in your code



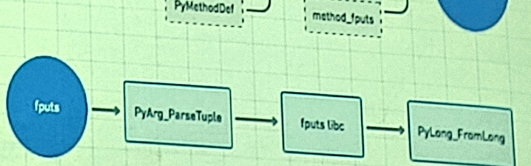

```
from distutils.core import setup, Extension

def main():
    setup(name="fputs",
          version="1.0.0",
          description="Python interface for the fputs C library function",
          author="<your name>",
          author_email="your_email@gmail.com",
          ext_modules=[Extension("fputs", ["fputsmodule.c"])])

if __name__ == "__main__":
    main()
```

Indian Institute of T

python_setup.py install

- Use in your code

```
>>> import fputs
>>> fputs.__doc__
'Python interface for the fputs C library function'
>>> fputs.__name__
'fputs'
>>> # Write to an empty file named `write.txt`
>>> fputs.fputs("Real Python!", "write.txt")
13
>>> with open("write.txt", "r") as f:
>>>     print(f.read())
'Real Python!'
```




```
from distutils.core import setup, Extension

def main():
    setup(name="fputs",
          version="1.0.0",
          description="Python interface for the fputs C library function",
          author="<your name>",
          author_email="your_email@gmail.com",
          ext_modules=[Extension("fputs", ["fputsmodule.c"])])

if __name__ == "__main__":
    main()
```

Accelerate matrixMultiply

```
#include <x86intrin.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <mm_malloc.h>

unsigned long long get_timestamp ()
{
    struct timeval now;
    gettimeofday (&now, NULL);
    return  now.tv_usec + (unsigned long long)now.tv_sec * 1000000;
}
```

```
void dgemm_avx256(const uint32_t n, const double* A, const double* B,
{

    for( uint32_t i = 0; i < n; i += 4 )
    {
        for( uint32_t j = 0; j < n; j++ )
        {
            __m256d c0 = _mm256_load_pd(C + i + j * n); /* c0 = C[i]
            for( uint32_t k = 0; k < n; k++ )
            {
                c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
                        _mm256_mul_pd( _mm256_load_pd(A + i + k * n
;
            }

            _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
        }
    }
}
```

Accelerate matrixMultiply

```
n.h>
.h>

h>

et_timestamp ()

w;
w, NULL);
ec + (unsigned long long)now.tv_sec * 1000000;
```

```
void dgemm_avx256(const uint32_t n, const double* A, const double* B, double* C)
{

    for( uint32_t i = 0; i < n; i += 4 )
    {
        for( uint32_t j = 0; j < n; j++ )
        {
            __m256d c0 = _mm256_load_pd(C + i + j * n); /* c0 = C[i][j] */
            for( uint32_t k = 0; k < n; k++ )
            {
                c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
                        _mm256_mul_pd( _mm256_load_pd(A + i + k * n), _mm256_broadcast_sd(B + k + j * n) ) )
;
            }

            _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
        }
    }
}
```

```
(long long)now.tv_sec * 1000000;
                    _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
                }
            }
        }


        void dgemm_avx512(const uint32_t n, const double* A, const double* B, double* C)
        {
            for( uint32_t i = 0; i < n; i += 8)
            {
                for( uint32_t j = 0; j < n; ++j)
                {
                    __m512d c0 = _mm512_load_pd(C + i + j * n); // c0 = C[i][j]
                    for( uint32_t k = 0; k < n; k++)
                    {
                        // c0 += A[i][k] * B[k][j]
                        __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B + j * n + k));
                        c0 = _mm512_fmadd_pd(_mm512_load_pd(A + n * k + i), bb, c0);
                    }
                    _mm512_store_pd(C + i + j * n, c0); // C[i][j] = c0
                }
            }
        }
```

Accelerate matrixMultiply


```
#include <x86intrin.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <mm_malloc.h>

unsigned long long get_timestamp ()
{
    struct timeval now;
    gettimeofday (&now, NULL);
    return  now.tv_usec + (unsigned long long)now.tv_sec * 1000000;
}
```

```
void dgemm_avx256(const uint32_t n, const double* A, const double* B, double* C)
{
    for( uint32_t i = 0; i < n; i += 4 )
    {
        for( uint32_t j = 0; j < n; j++ )
        {
            __m256d c0 = _mm256_load_pd(C + i + j * n); /* c0 = C[i][j] */
            for( uint32_t k = 0; k < n; k++ )
            {
                c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
                        _mm256_mul_pd( _mm256_load_pd(A + i + k * n), _mm256_broadcast_sd(B + k + j * n) ) )
;
            }

            _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
        }
    }
}
```

```
void dgemm_avx512(const uint32_t n, const double* A, const double* B, double* C)
{
    for( uint32_t i = 0; i < n; i += 8)
    {
        for( uint32_t j = 0; j < n; ++j)
        {
            __m512d c0 = _mm512_load_pd(C + i + j * n); // c0 = C[i][j]
            for( uint32_t k = 0; k < n; k++)
            {
                // c0 += A[i][k] * B[k][j]
                __m512d bb = _mm512_broadcastsd_pd( _mm_load_sd(B + j * n + k));
                c0 = _mm512_fmadd_pd( _mm512_load_pd(A + n * k + i), bb, c0);
            }
            _mm512_store_pd(C + i + j * n, c0); // C[i][j] = c0
        }
    }
}
```

```
void dgemm_basic(const uint32_t n, const double* A, const double* B, double* C)
{
    for(uint32_t i = 0; i < n; ++i)
    {
        for(uint32_t j = 0; j < n; ++j)
        {
            double cij = C[i + j * n]; /* cij = C[i][j] */
            for(uint32_t k = 0; k < n; k++)
            {
                cij += A[i + k * n] * B[k + j * n]; /* cij += A[i][k]*B[k][j] */
            }
            C[i + j * n] = cij; /* C[i][j] = cij */
        }
    }
}
```

```
void dgemm_avx512(const uint32_t n, const double
{
    for( uint32_t i = 0; i < n; i += 8)
    {
        for( uint32_t j = 0; j < n; ++j)

            __m512d c0 = _mm512_load_pd(C + i +
            for( uint32_t k = 0; k < n; k++)
            {
                // c0 += A[i][k] * B[k][j]
                __m512d bb = _mm512_broadcastsd
                c0 = _mm512_fmadd_pd(_mm512_loa
            }
            _mm512_store_pd(C + i + j * n, c0)
```

Accelerate matrixMultiply


```
int main(){
        uint32_t trial_no = 11;
        uint32_t n = 32 * 20;

        double *a;
        double *b;
        double *c;
        unsigned long long t0;
        unsigned long long t1;
        unsigned long long t;
struct timeval  tv1, tv2;


double result;

a = (double*) _mm_malloc (n * n * sizeof(double), 64);
b = (double*) _mm_malloc (n * n * sizeof(double), 64);
c = (double*) _mm_malloc (n * n * sizeof(double), 64);


  for(uint32_t i = 0; i < n * n; ++i)
  {
          a[i] = rand();
  }
```

```
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_basic(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv
}
result =  result / (double)(trial_no);
printf("Time Taken: Ref:\t%f\n",result);
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double)
 }
result =  result / (double)(trial_no);
printf("Time Taken: AVX256:\t%f\n",result);
```

Accelerate matrixMultiply

```
int main(){
        uint32_t trial_no = 11;
        uint32_t n = 32 * 20;

        double *a;
        double *b;
        double *c;
        unsigned long long t0;
        unsigned long long t1;
        unsigned long long t;
uct timeval  tv1, tv2;

le result;

double*) _mm_malloc (n * n * sizeof(double), 64);
double*) _mm_malloc (n * n * sizeof(double), 64);
double*) _mm_malloc (n * n * sizeof(double), 64);

(uint32_t i = 0; i < n * n; ++i)

    a[i] = rand();
```

```
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_basic(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: Ref:\t%f\n",result);
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: AVX256:\t%f\n",result);
```

```
double*) _mm_malloc (n * n * sizeof(double), 64);
double*) _mm_malloc (n * n * sizeof(double), 64);
double*) _mm_malloc (n * n * sizeof(double), 64);

(uint32_t i = 0; i < n * n; ++i)

    a[i] = rand();
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{
        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: AVX256:\t%f\n",result);
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{
        gettimeofday(&tv1, NULL);
        dgemm_avx512(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: AVX512:\t%f\n",result);
```

```
Wall -pedantic matmul-3.c  -march=icelake-server
```

```c
for(uint32_t i = 0; i < n * n; ++i)
{
        a[i] = rand();
}
```

```c
        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
    }
    result =  result / (double)(trial_no);
    printf("Time Taken: AVX256:\t%f\n",result);
```

```
kolin@minerva:                    $ ./a.out
Time Taken: Ref:          1.027594
Time Taken: AVX256:       0.436642
Time Taken: AVX512:       0.412659
```

```c
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{
        gettimeofday(&tv1, NULL);
        dgemm_avx512(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
}
result =  result / (double)(trial_no);
printf("Time Taken: AVX512:\t%f\n",result);
```

```
gcc -Wall -pedantic matmul-3.c -march=icelake-server
```

Indian Institute

Accelerate matrixMultiply

```
int main(){
        uint32_t trial_no = 11;
        uint32_t n = 32 * 20;

        double *a;
        double *b;
        double *c;
        unsigned long long t0;
        unsigned long long t1;
        unsigned long long t;
    struct timeval  tv1, tv2;


    double result;

    a = (double*) _mm_malloc (n * n * sizeof(double), 64);
    b = (double*) _mm_malloc (n * n * sizeof(double), 64);
    c = (double*) _mm_malloc (n * n * sizeof(double), 64);


        for(uint32_t i = 0; i < n * n; ++i)
        {
                a[i] = rand();
        }
```

```
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_basic(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
 result =  result / (double)(trial_no);
 printf("Time Taken: Ref:\t%f\n",result);
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
 }
 result =  result / (double)(trial_no);
 printf("Time Taken: AVX256:\t%f\n",result);
```

```
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{
        gettimeofday(&tv1, NULL);
        dgemm_avx512(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
 result =  result / (double)(trial_no);
 printf("Time Taken: AVX512:\t%f\n",result);
```

```
kolin@minerva: /accel/extension/two$ ./a.out
Time Taken: Ref:        1.027594
Time Taken: AVX256:     0.436642
Time Taken: AVX512:     0.412659
```

```
gcc -Wall -pedantic matmul-3.c  -march=icelake-server
```

Using shared Objects

```
from ctypes import CDLL, POINTER
from ctypes import c_size_t, c_double, c_uint
import numpy as np
from numpy import random
import time

# load the library
mylib = CDLL("/home/kolin/accel/extension/two/mylib.so")

# C-type corresponding to numpy array
ND_POINTER_1 = np.ctypeslib.ndpointer(dtype=np.float64,
                                      ndim=1,
                                      flags="C")

# define prototypes
mylib.print_array.argtypes = [ND_POINTER_1 , c_size_t]
mylib.print_array.restype = None
mylib.dgemm_basic.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1 ]
mylib.dgemm_basic.restype = None
mylib.dgemm_avx512.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
mylib.dgemm_avx512.restype = None
mylib.dgemm_avx256.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
mylib.dgemm_avx256.restype = None
```

```
gcc -Wall -pedantic -shared -fPIC -o mylib.so try.c  -march=icelake-server
```

```
/Auto/accel/extension/two/mylib.so")

ype corresponding to numpy array
INTER_1 = np.ctypeslib.ndpointer(dtype=np.float64,
                                 ndim=1,
                                 flags="C")

e prototypes
rint_array.argtypes = [ND_POINTER_1 , c_size_t]
rint_array.restype = None
gemm_basic.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1 ]
gemm_basic.restype = None
emm_avx512.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
emm_avx512.restype = None
emm_avx256.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
mm_avx256.restype = None
```

```
-o mylib.so try.c  -march=icelake-server
```

```
n=32*20

#s = (2,2)
s=n*n
a = random.rand(s)
b =random.rand(s)
c =np.zeros(s)
X=np.ones(5)

# call function
mylib.print_array(X, X.size)
start=time.time()
mylib.dgemm_basic(n,a,b,c)
end=time.time()
print(c)
print("Elapsed Time Ref:", end-start)
c =np.zeros(s)
start1=time.time()
mylib.dgemm_avx256(n,a,b,c)
end1=time.time()
print("Elapsed Time AVX256:", end1-start1)
```

Graphical Pipeline:

- Set of steps need to be taken in order to turn 3D scene into 2D image
  - The graphical pipeline is conceptual model
  - Highly dependent on the underlying available Software and Hardware accelerators

Graphical Pipeline:

- Set of steps need to be taken in order to turn 3D scene into 2D image
  - The graphical pipeline is conceptual model
  - Highly dependent on the underlying available Software and Hardware accelerators
- The model of graphical pipeline is usually used in real-time rendering
  - Each step is backed with efficient algorithms that are usually hardware accelerated (e.g., using GPUs)

image ... to turn 3D scene into 2D

- The graphical pipeline is conceptual model
- Highly dependent on the underlying available Software and Hardware accelerators

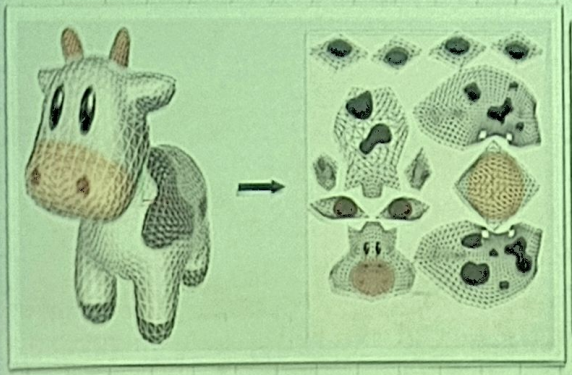
- The model of graphical pipeline is usually used in real-time rendering
  - Each step is backed with efficient algorithms that are usually hardware accelerated (e.g., using GPUs)
- How do you represent 3D surfaces ?
- A 3D mesh (usually triangles):
  - Simple objects can be composed of thousands of triangles!

Graphical Pipeline:

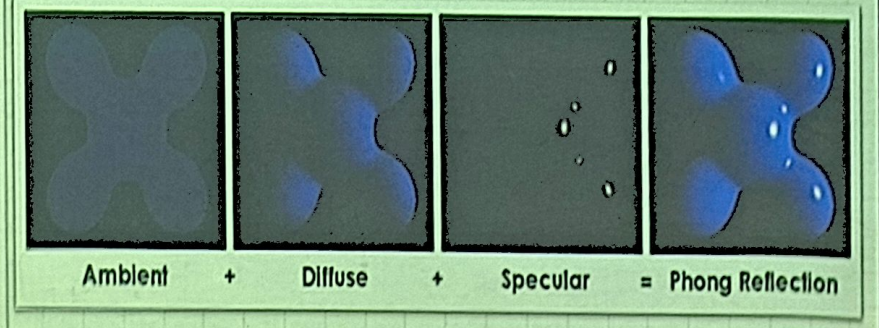
- Set of steps need to be taken in order to turn 3D scene into 2D image
  - The graphical pipeline is conceptual model
  - Highly dependent on the underlying available Software and Hardware accelerators
- The model of graphical pipeline is usually used in real-time rendering
  - Each step is backed with efficient algorithms that are usually hardware accelerated (e.g., using GPUs)
- How do you represent 3D surfaces ?
- A 3D mesh (usually triangles):
  - Simple objects can be composed of thousands of triangles!

Graphic Processing – Some History

- 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- 3D graphics processing is immensely computation-intensive

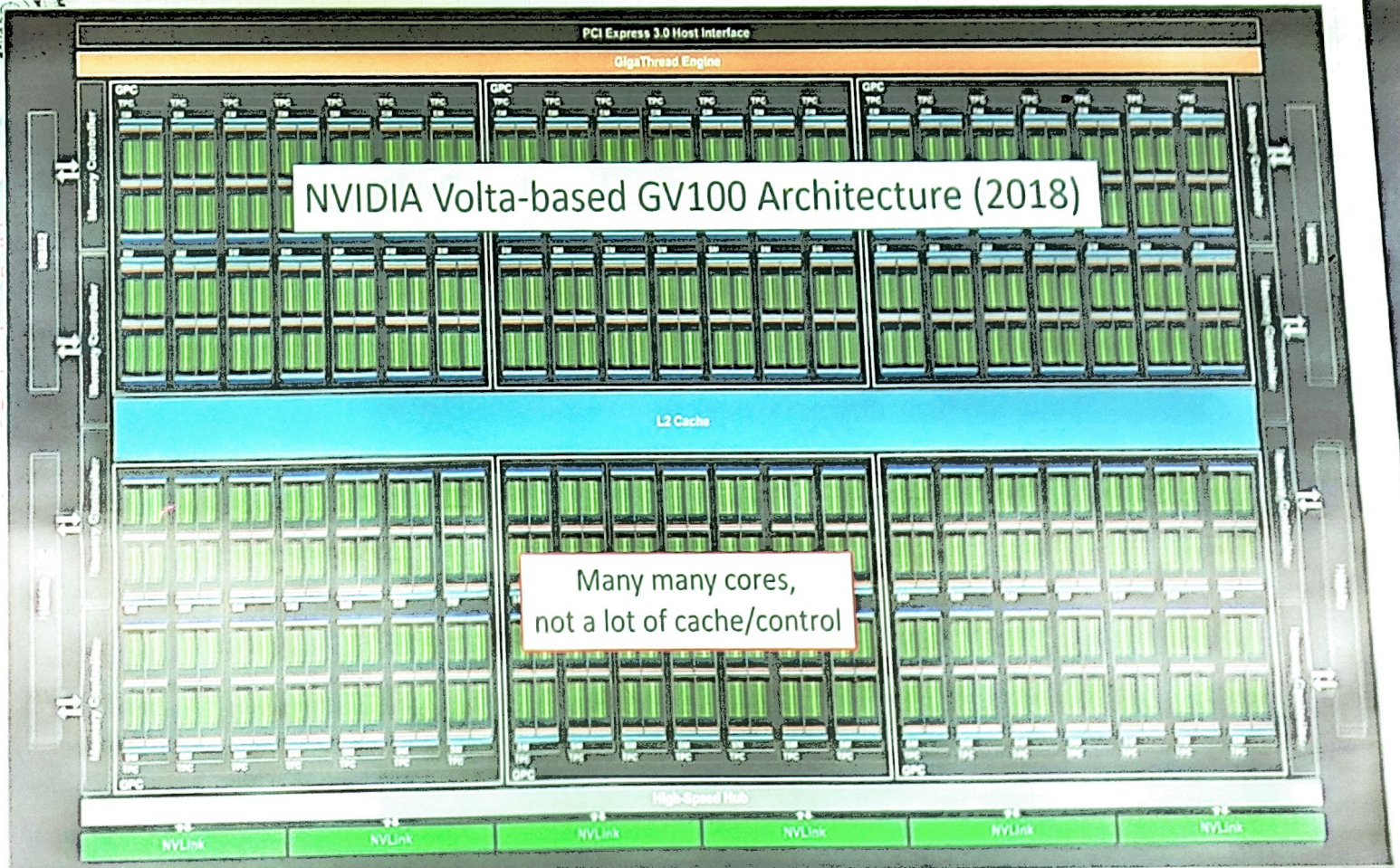


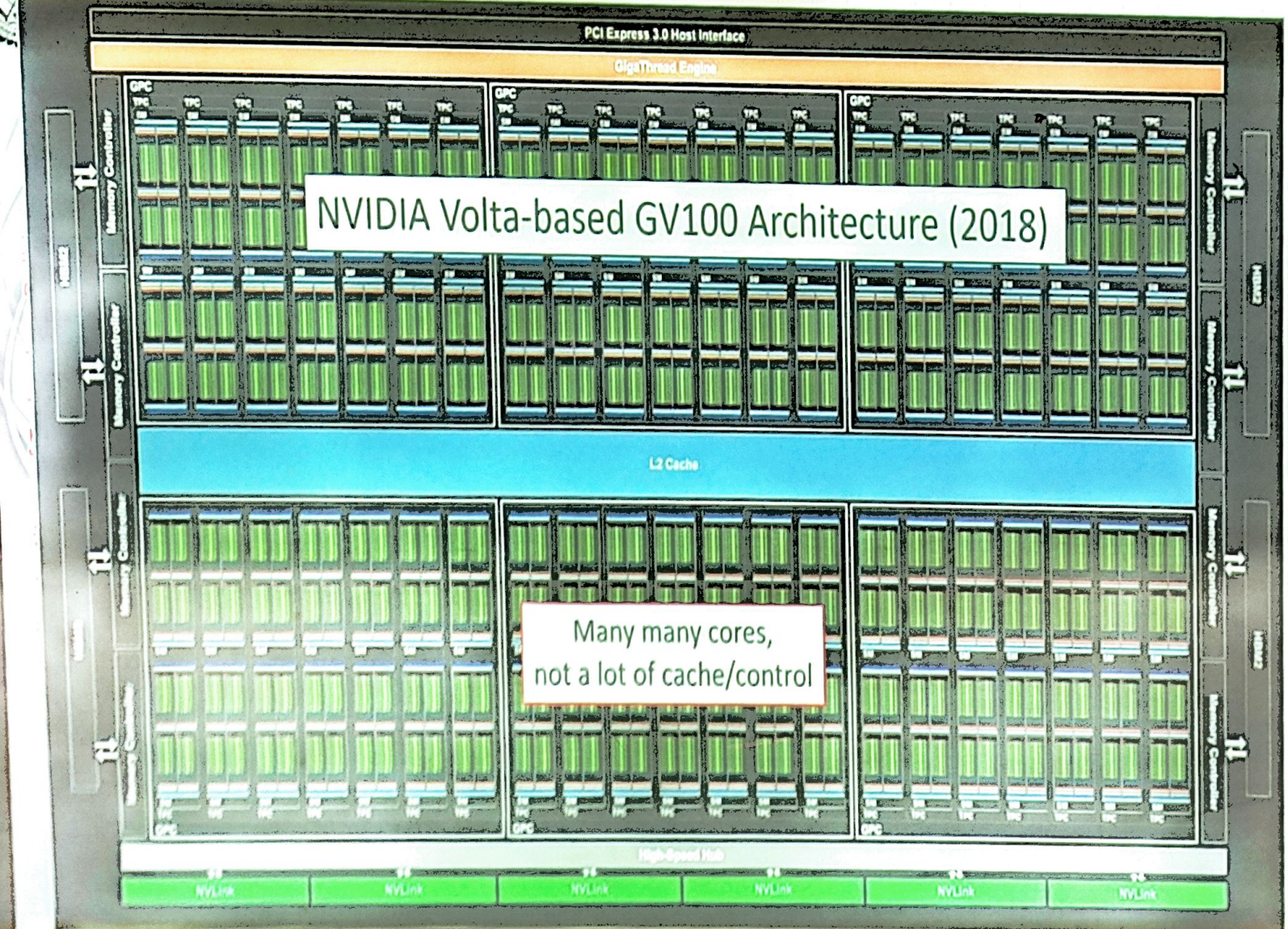
Texture mapping



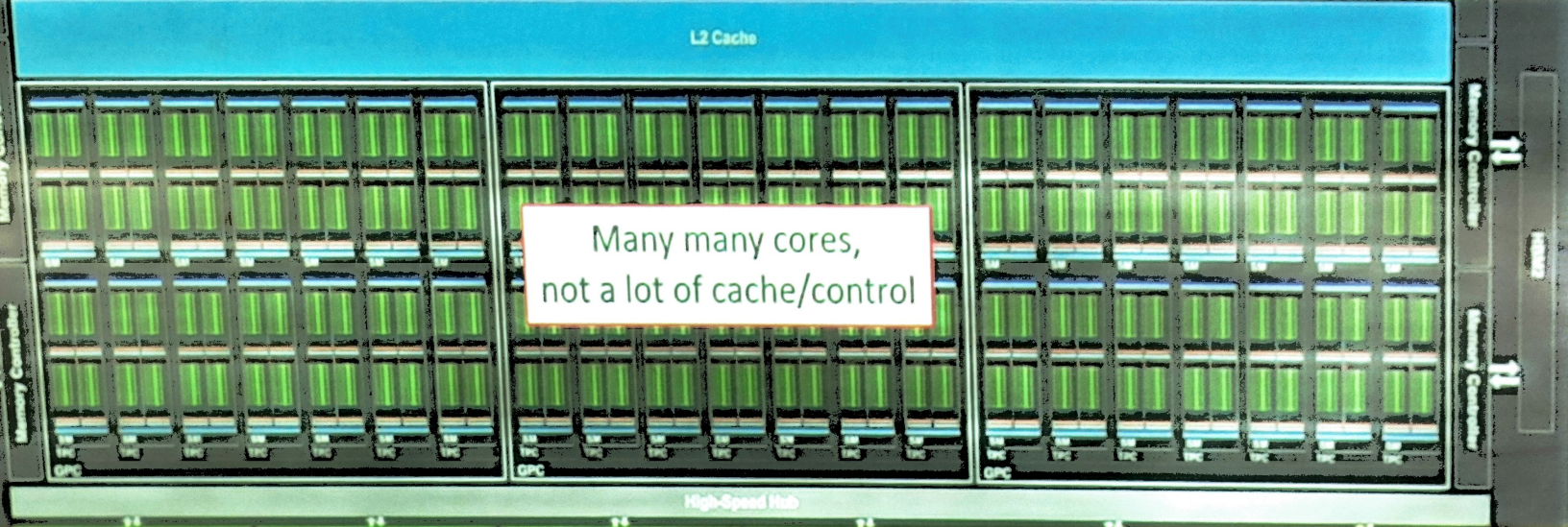
Shading

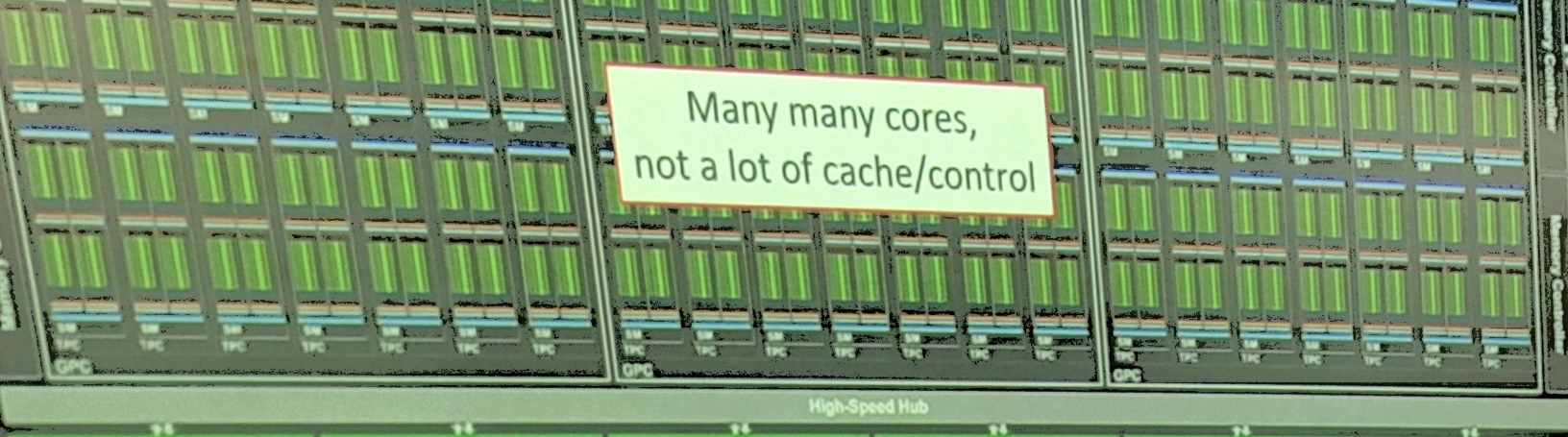
Warren Moore, "Textures and Samplers in Metal," Metal by Example, 2014


PCI Express 3.0 Host Interface
GigaThread Engine
NVIDIA Volta-based GV100 Architecture (2018)
L2 Cache
Many many cores,
not a lot of cache/control
High-Speed Hub
NVLink
NVLink
NVLink
NVLink
NVLink
NVLink

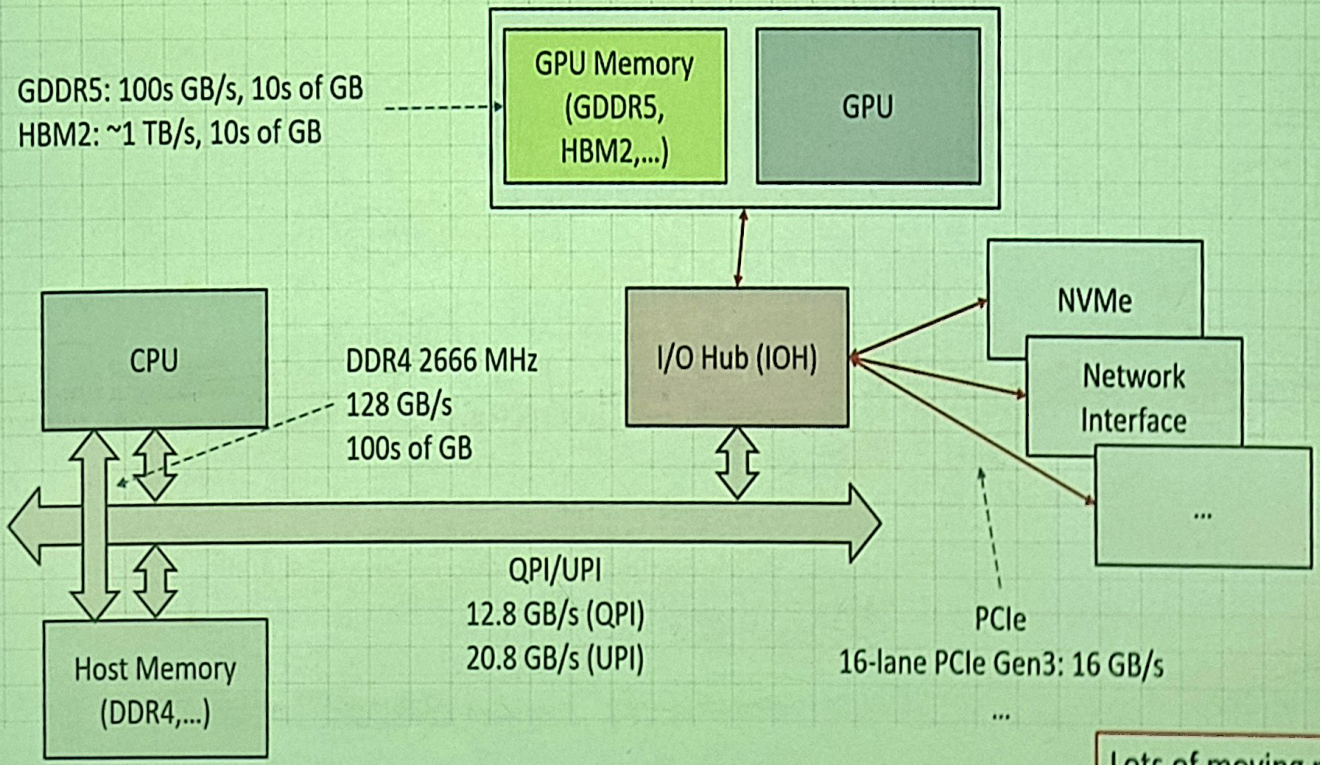

PCI Express 3.0 Host Interface
GigaThread Engine
NVIDIA Volta-based GV100 Architecture (2018)
L2 Cache
Many many cores,
not a lot of cache/control
NVLink


GigaThread Engine
GPC
TPC
SM
Memory Controller
L2 Cache
NVIDIA Volta-based GV100 Architecture (2018)


L2 Cache
Many many cores,
not a lot of cache/control
GPC
GPC
GPC
High-Speed Hub

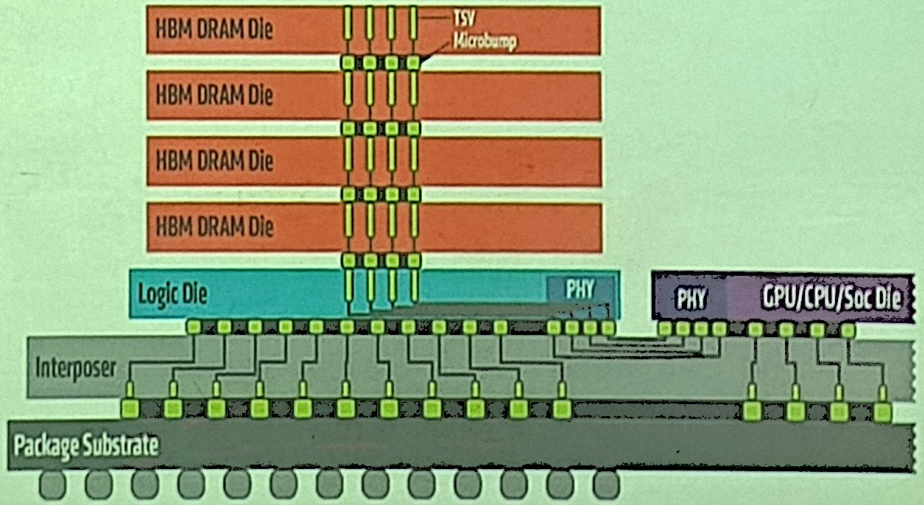

Many many cores,
not a lot of cache/control
High-Speed Hub

System Architecture Snapshot With a GPU



High-Performance Graphics Memory

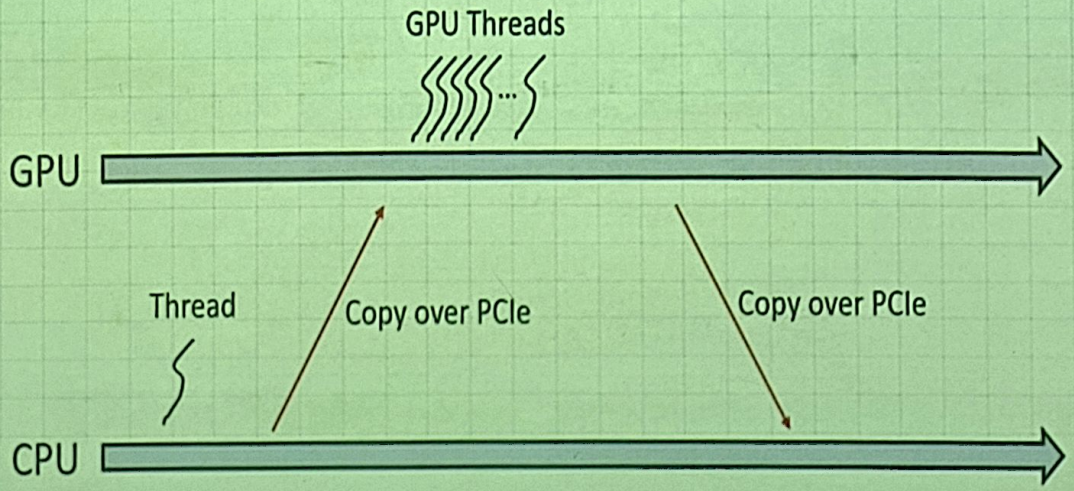
- Modern GPUs even employing 3D-stacked memory via silicon interposer
  - Very wide bus, very high bandwidth
  - e.g., HBM2 in Volta

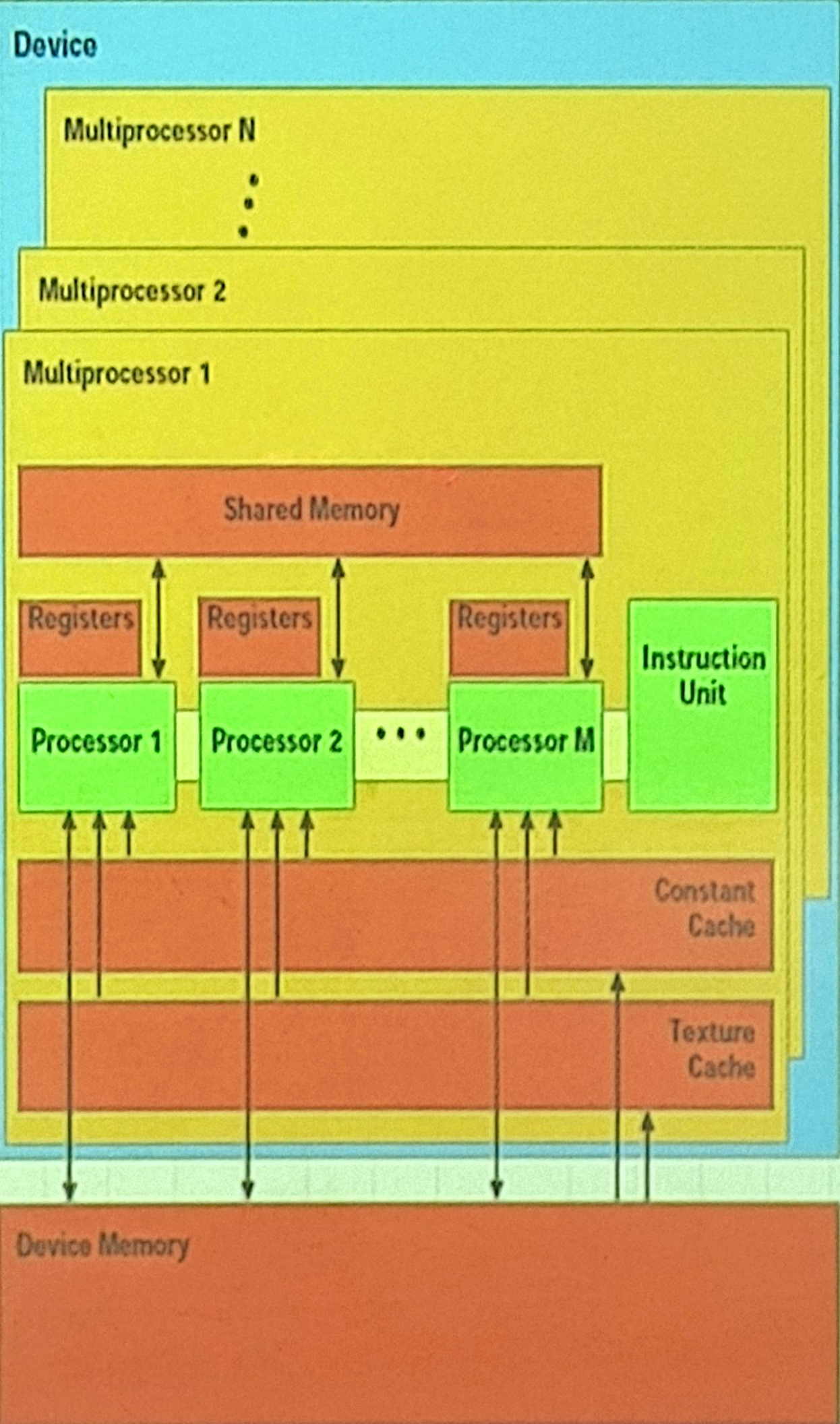


Graphics Card Hub, "GDDR5 vs GDDR5X vs HBM vs HBM2 vs GDDR6 Memory Comparison," 2019

Massively Parallel Architecture For Massively Parallel Workloads!

- NVIDIA CUDA (Compute Uniform Device Architecture) – 2007
  - A way to run custom programs on the massively parallel architecture!
- OpenCL specification released – 2008
- Both platforms expose synchronous execution of a massive number of threads




Device
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
Shared Memory
Registers
Registers
Registers
Instruction Unit
Processor 1
Processor 2
Processor M
Constant Cache
Texture Cache
Device Memory

NVIDIA GPU Architecture

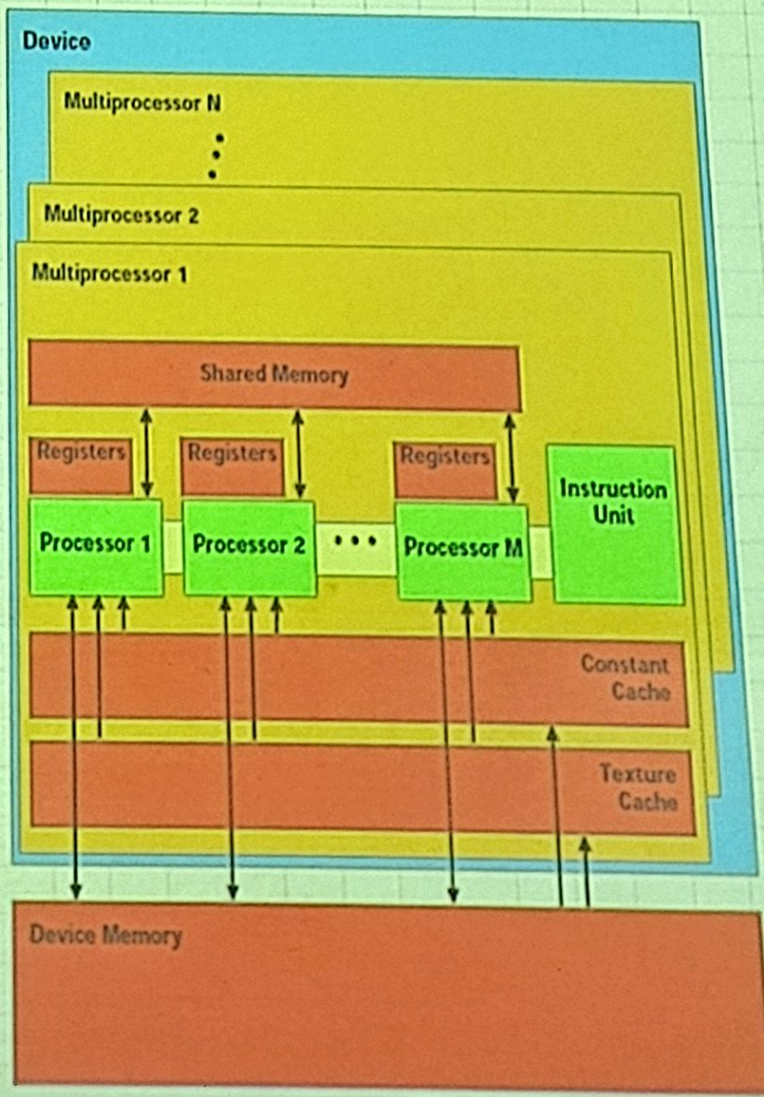


Figure is courtesy of NV

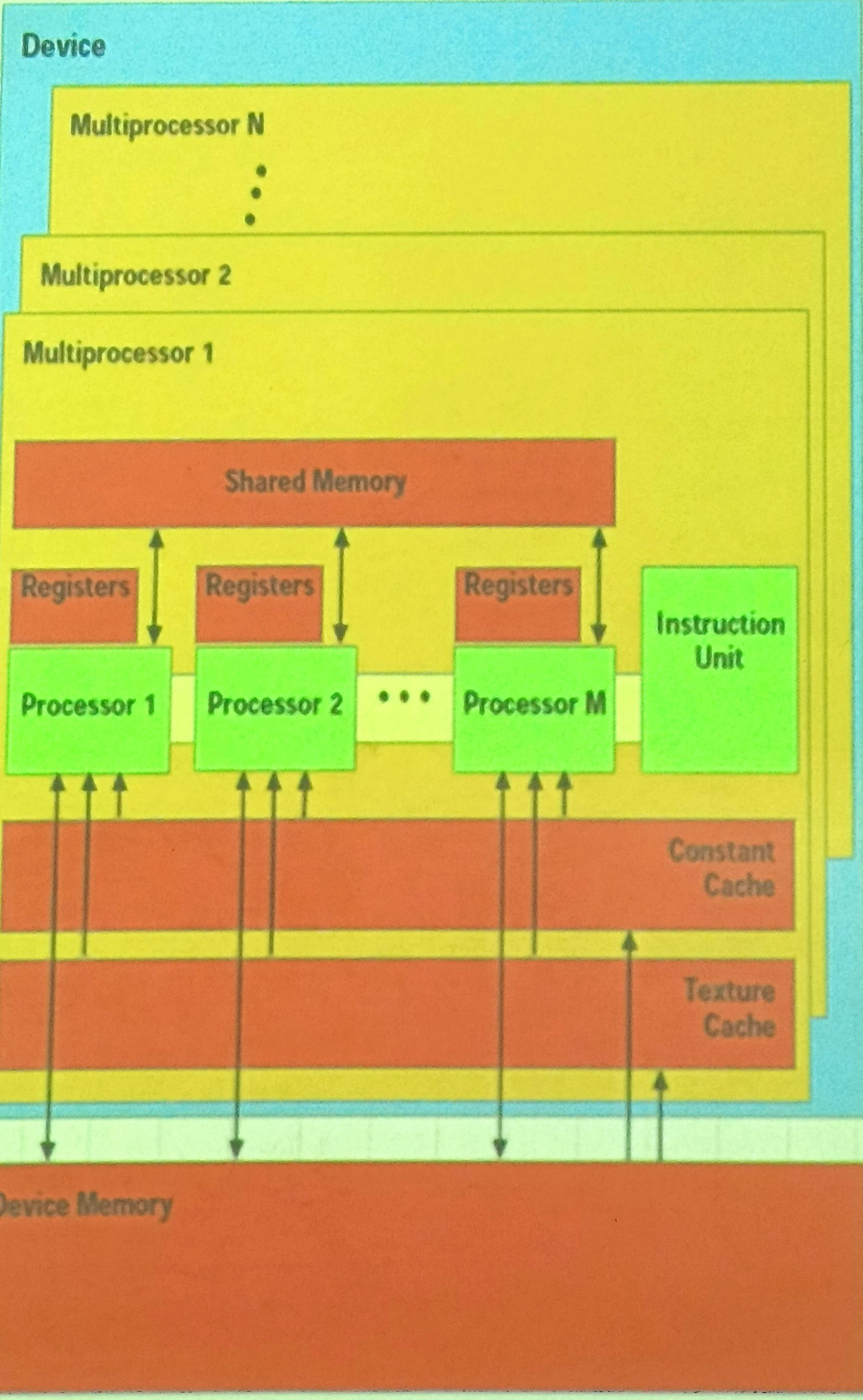


Figure is courtesy of NVI

NVIDIA GPU Architecture

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals

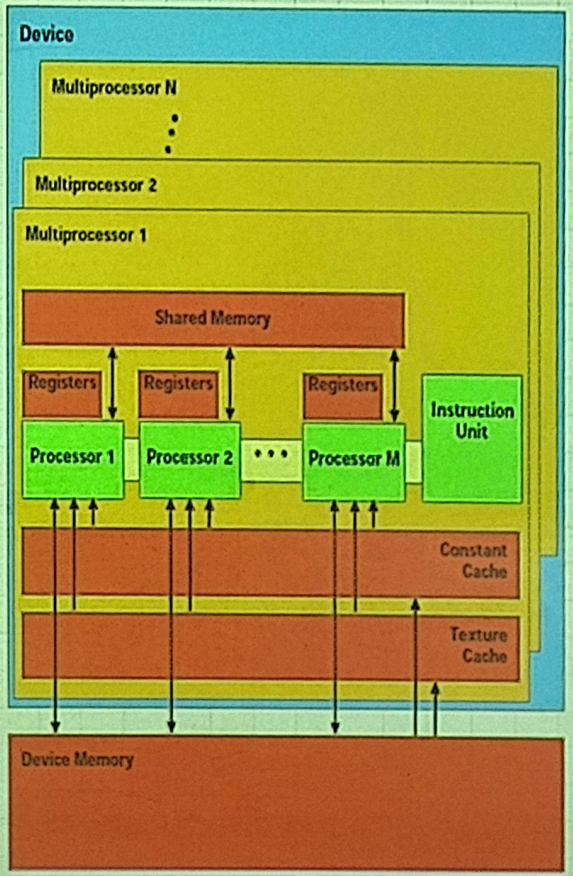


Figure is courtesy of NVIDIA

NVIDIA GPU Architecture

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals
  - A multithreaded instruction unit
  - On-chip shared memory

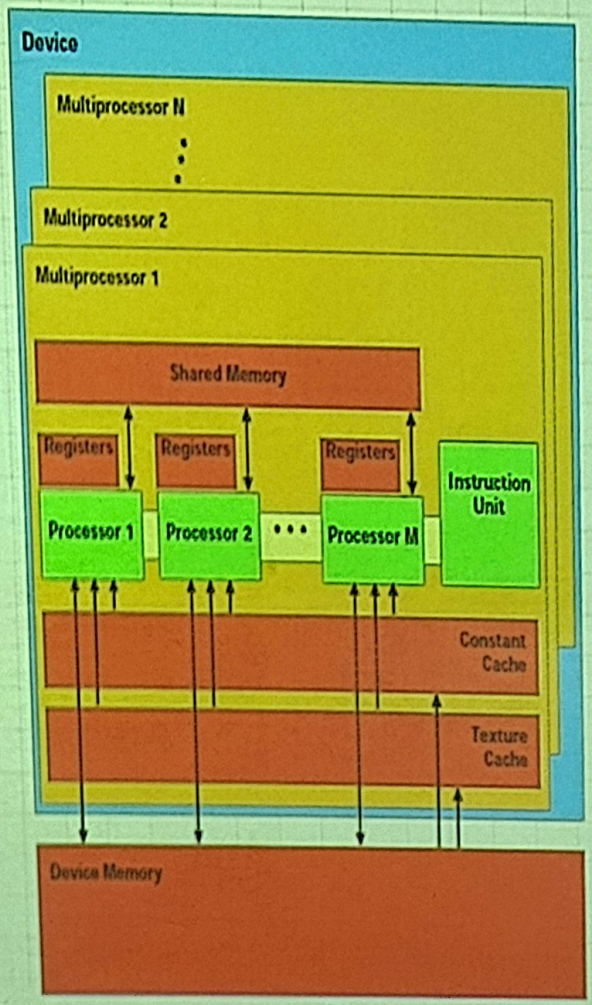
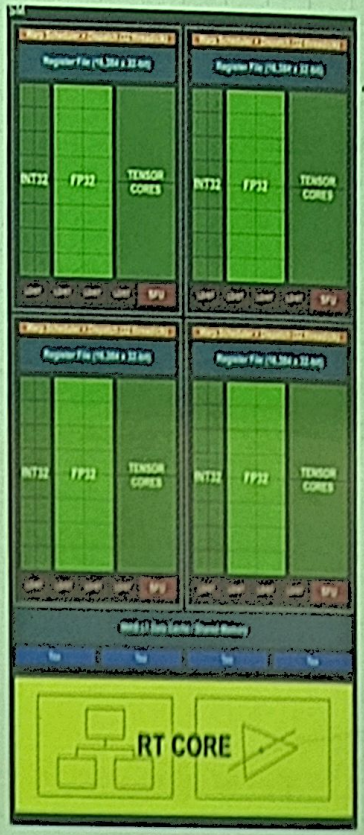


Figure is courtesy of NVIDIA

Volta Execution Architecture

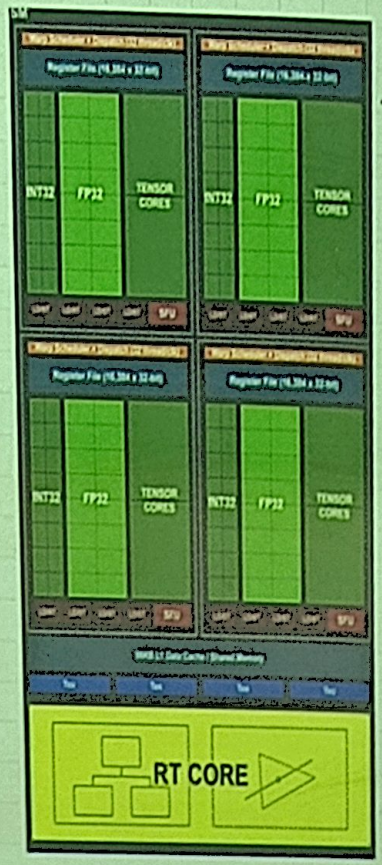
- 64 INT32 Cores, 64 FP32 Cores, 4 Tensor Cores, Ray-tracing cores..
  - Specialization to make use of chip space...?
- Not much on-chip memory per thread
  - 96 KB Shared memory
  - 1024 Registers per FP32 core
- Hard limit on compute management
  - 32 blocks AND 2048 threads AND 1024 threads/block
  - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
  - Enough registers/shared memory for all threads must be available (all context is resident during execution)



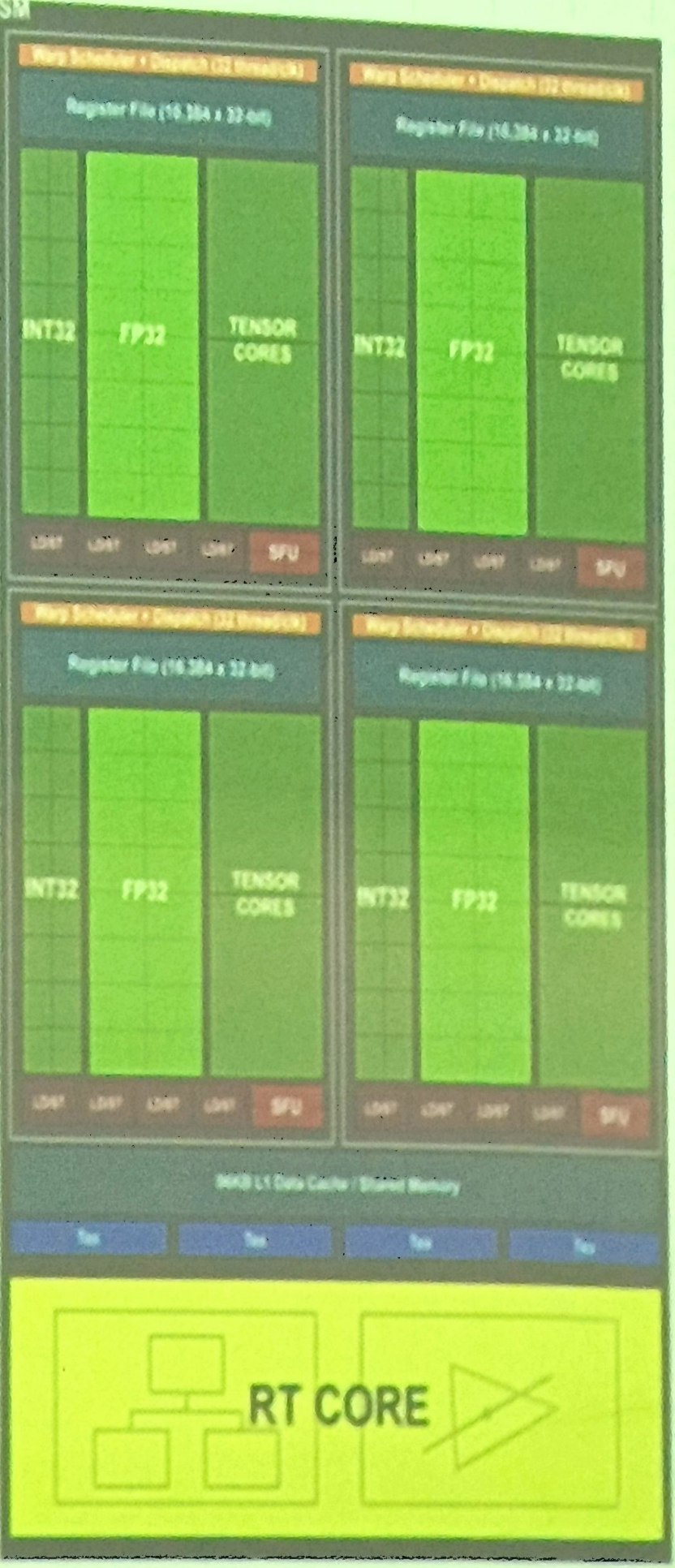
More threads than cores = Threads interleaved to hide memory latency

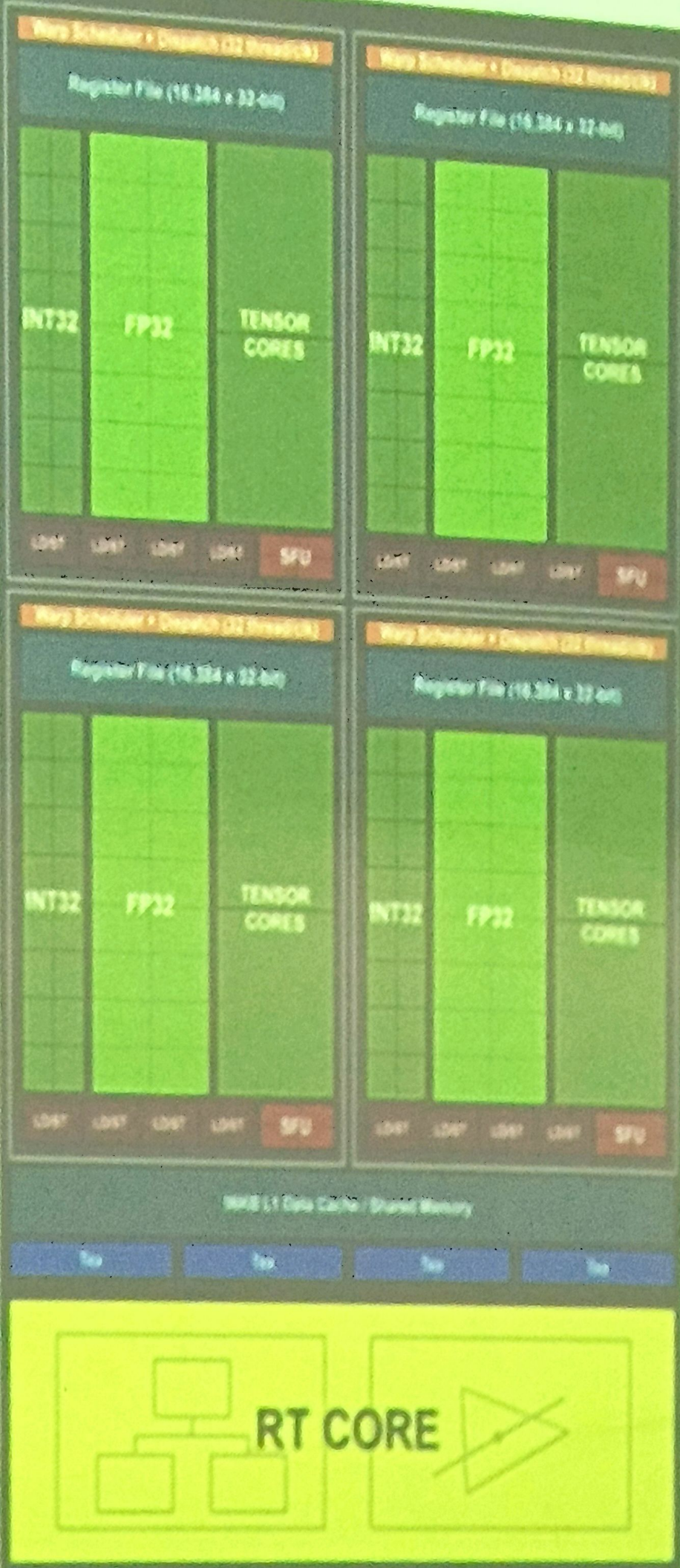
Volta Execution Architecture

- 64 INT32 Cores, 64 FP32 Cores, 4 Tensor Cores, Ray-tracing cores..
  - Specialization to make use of chip space...?
- Not much on-chip memory per thread
  - 96 KB Shared memory
  - 1024 Registers per FP32 core
- Hard limit on compute management
  - 32 blocks AND 2048 threads AND 1024 threads/block
  - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
  - Enough registers/shared memory for all threads must be available (all context is resident during execution)



More threads than cores – Threads interleaved to hide memory latency


SM
INT32
FP32
TENSOR CORES
SFU
RT CORE


INT32
FP32
TENSOR CORES
SFU
INT32
FP32
TENSOR CORES
SFU
INT32
FP32
TENSOR CORES
SFU
INT32
FP32
TENSOR CORES
SFU
RT CORE

Resource Balancing Details

- How many threads in a block?
- Too small: 4x4 window == 16 threads
  - 128 blocks to fill 2048 thread/SM
  - SM only supports 32 blocks -> only 512 threads used
    - SM has only 64 cores… does it matter? Sometimes!
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
- Too large: 1024 threads using more than 64 registers
- Limitations vary across platforms (Fermi, Pascal, Volta, …)

Resource Balancing Details

- How many threads in a block?
- Too small: 4x4 window == 16 threads
  - 128 blocks to fill 2048 thread/SM
  - SM only supports 32 blocks -> only 512 threads used
    - SM has only 64 cores... does it matter? Sometimes!
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
- Too large: 1024 threads using more than 64 registers
- Limitations vary across platforms (Fermi, Pascal, Volta, ...)

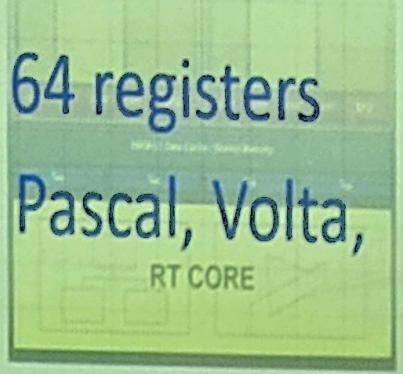
Volta Execution Architecture

- 64 INT32 Cores, 64 FP32 Cores, 4 Tensor Cores, Ray-tracing cores..
  - Specialization to make use of chip space...?
- Not much on-chip memory per thread
  - 96 KB Shared memory
  - 1024 Registers per FP32 core
- Hard limit on compute management
  - 32 blocks AND 2048 threads AND 1024 threads/block
  - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
  - Enough registers/shared memory for all threads must be available (all context is resident during execution)

More threads than cores – Threads interleaved to hide memory latency

Resource Balancing Details

- How many threads in a block?
- Too small: 4x4 window == 16 threads
  - 128 blocks to fill 2048 thread/SM
  - SM only supports 32 blocks -> only 512 threads used
    - SM has only 64 cores... does it matter? Sometimes!
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
- Too large: 1024 threads using more than 64 registers
- Limitations vary across platforms (Fermi, Pascal, Volta, ...)



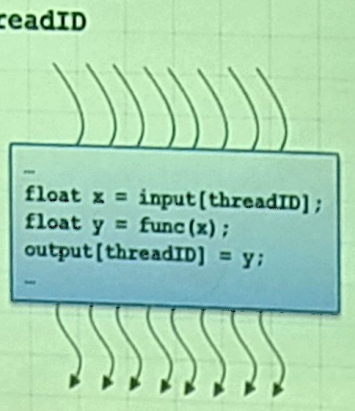
More threads than cores – Threads interleaved to hide memory latency

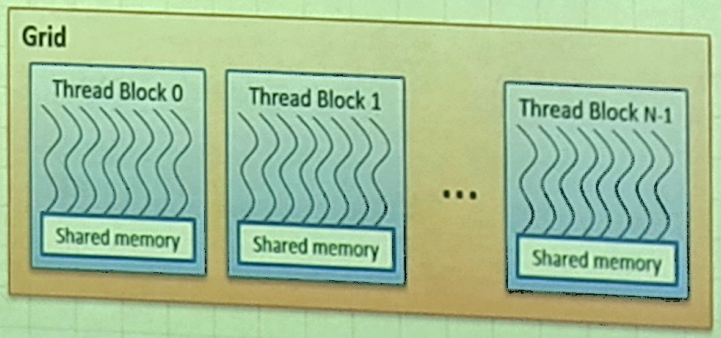
Warp Scheduling Unit

- Threads in a block are executed in 32-thread "warp" unit
  - Not part of language specs, just architecture specifics
  - A warp is SIMD – Same PC, same instructions executed on every core
- What happens when there is a conditional statement?
  - Prefix operations, or control divergence
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
  - More on this later!
- Too large: 1024 threads using more than 64 registers
- Warps have been 32-threads so far, but may change in the future
- Limitations vary across platforms (Fermi, Pascal, Volta, ...)

CUDA Programming Model

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions
- Threads are arranged as a grid of thread blocks
  - Threads within a block have access to a segment of shared memory

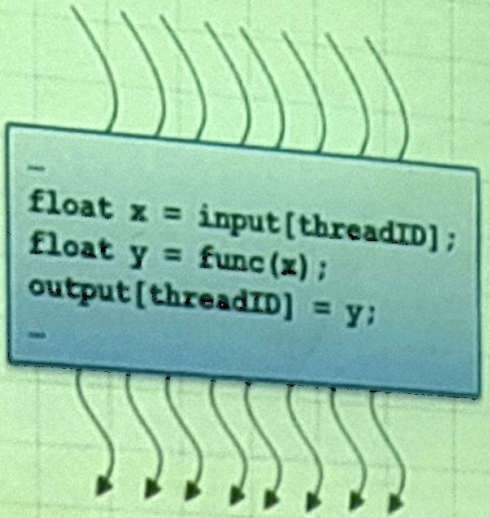


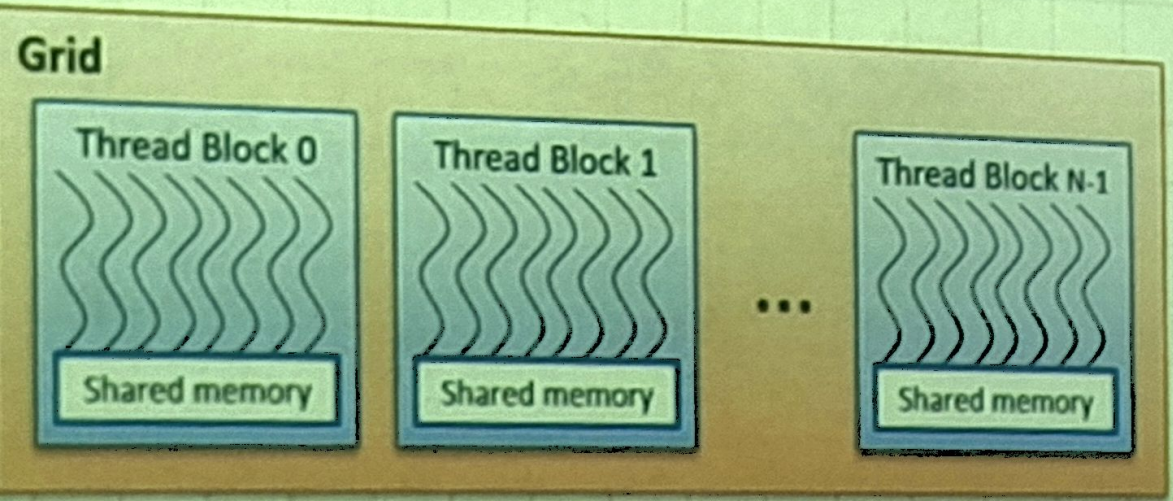


of thread blocks

make control decisions

threadID

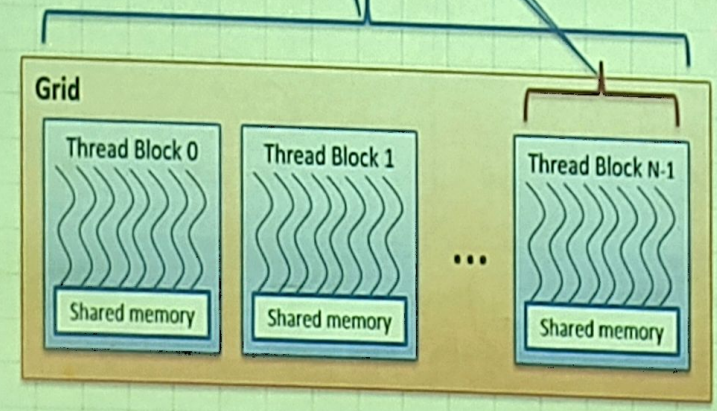




Kernel Invocation Syntax

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

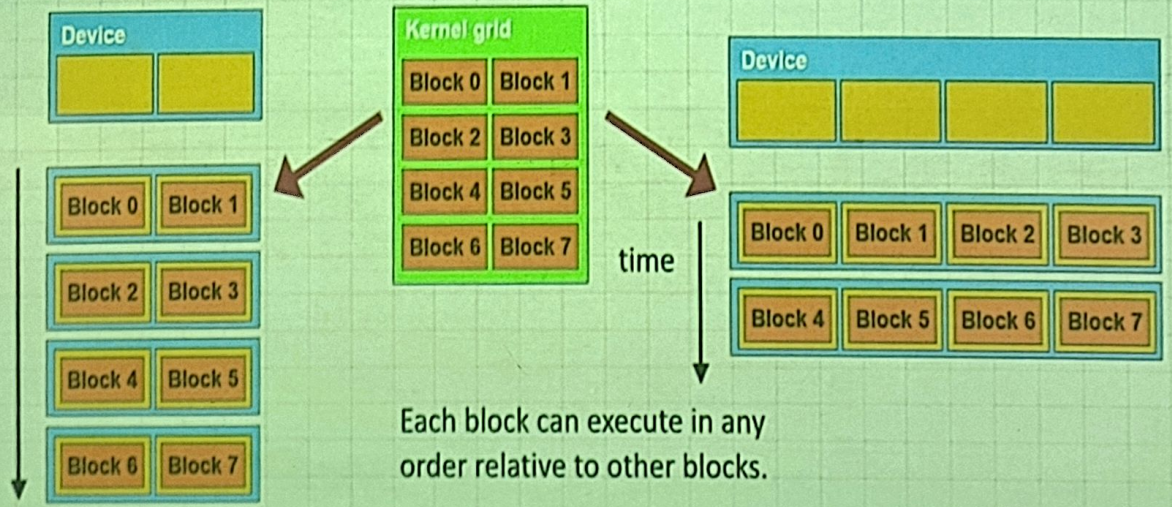
block ID within a grid

number of theards per block

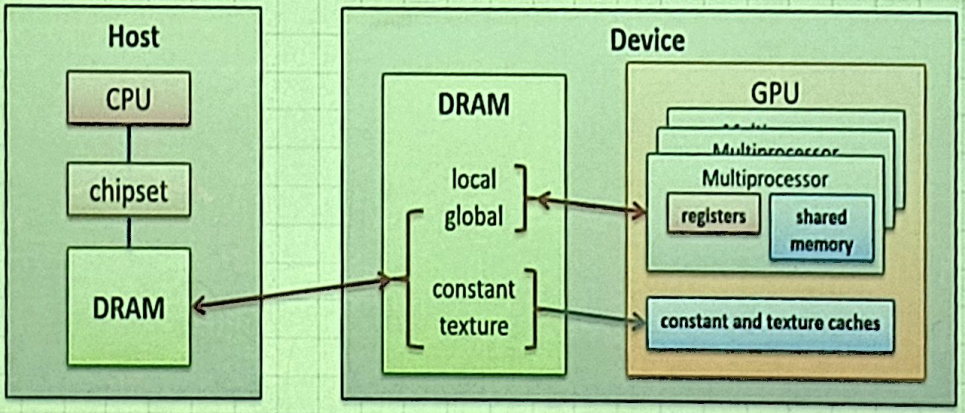
thread ID within a thread block

Mapping Threads to the Hardware

- Blocks of threads are transparently assigned to SMs
  - A block of threads executes on one SM & does not migrate
  - Several blocks can reside concurrently on one SM

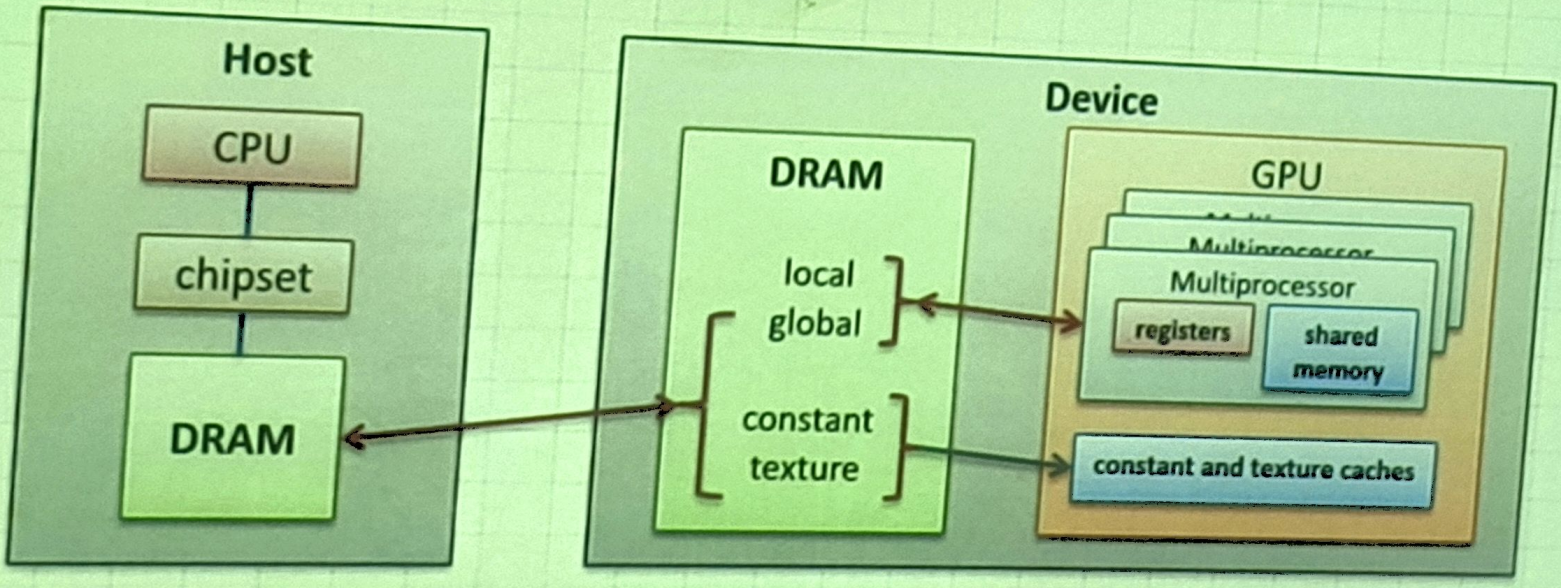


GPU Memory Hierarchy



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

CPU Memory Hierarchy



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

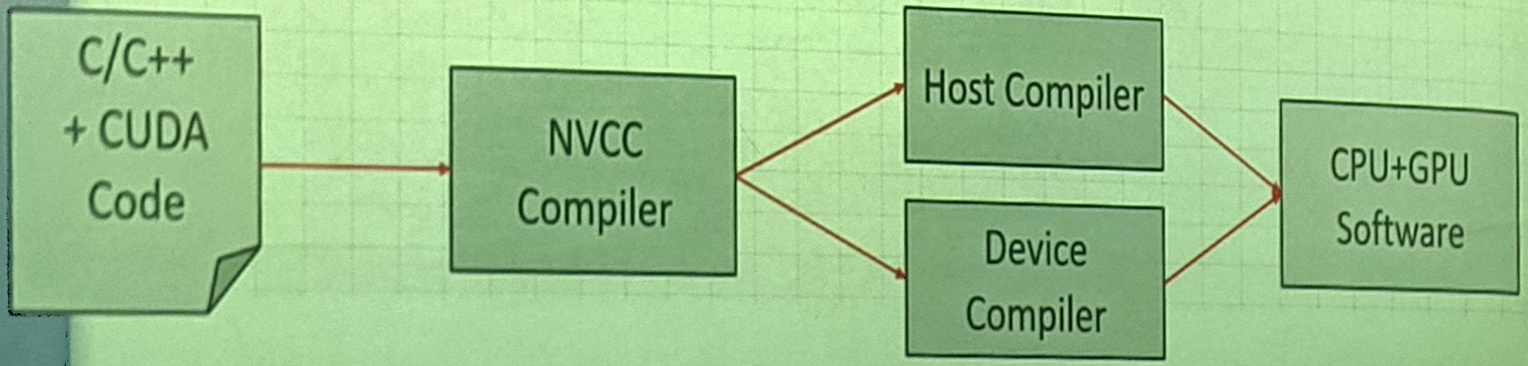
CPU side

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Asynchronous call

GPU side

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```



Simple CUDA Example

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

1 block

N threads per block

Should wait for kernel to finish

__global__:
In GPU, called from host/GPU
__device__:
In GPU, called from GPU
__host__:
In host, called from host

N instances of VecAdd spawned in GPU

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Only void allowed

Which of N threads am I?
See also: blockIdx

One function can be both

CPU-Only Version

```
void vecAdd(int N, float* A, float* B, float* C) {
   for (int i = 0; i < N; i++)  C[i] = A[i] + B[i];
}

int  main(int argc, char **argv)
{
   int N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   vecAdd(N, A, B, C);  // call compute kernel

   free(A); free(B); free(C);
}
```

Adding GPU support

```
int  main(int argc, char **argv)
{
   int  N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   float *devPtrA, *devPtrB, *devPtrC;

   cudaMalloc((void**)&devPtrA, N * sizeof(float));
   cudaMalloc((void**)&devPtrB, N * sizeof(float));
   cudaMalloc((void**)&devPtrC, N * sizeof(float));

   cudaMemcpy(devPtrA, A, N * sizeof(float),  cudaMemcpyHostToDevice);
   cudaMemcpy(devPtrB, B, N * sizeof(float),  cudaMemcpyHostToDevice);
```

Adding GPU support

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);

cudaMemcpy(C, devPtrC, N * sizeof(float),  cudaMemcpyDeviceToHost);

cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);

free(A); free(B); free(C);
}
```

GPU Kernel

- **CPU version**

```
void vecAdd(int N, float* A, float* B, float* C)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

```
cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);
```

Copy results from device memory to the host memory

```
free(A); free(B); free(C);
```

Device memory de-allocation

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

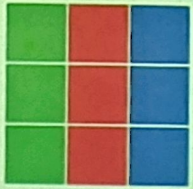
```
int main(int argc, char* argv[])
{
   int N = 1024;

   struct timeval t1, t2, ta, tb;
   long msec1, msec2;
   float flop, mflop, gflop;

   float *a = (float *)malloc(N*N*sizeof(float));
   float *b = (float *)malloc(N*N*sizeof(float));
   float *c = (float *)malloc(N*N*sizeof(float));

   minit(a, b, c, N);

   gettimeofday(&t1, NULL);
   mmult(a, b, c, N);  // a = b * c
   gettimeofday(&t2, NULL);

   mprint(a, N, 5);

   free(a);
   free(b);
   free(c);

   msec1 = t1.tv_sec * 1000000 + t1.tv_usec;
   msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
   msec2 -= msec1;
   flop = N*N*N*2.0f;
   mflop = flop / msec2;
   gflop = mflop / 1000.0f;
   printf("msec = %10ld   GFLOPS = %.3f\n", msec2, gflop);
}
```

```
// a = b * c
void mmult(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         for (int i = 0; i < N; i++)
            a[i+j*N] += b[i+k*N]*c[k+j*N];
}


void minit(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int i = 0; i < N; i++) {
         a[i+N*j] = 0.0f;
         b[i+N*j] = 1.0f;
         c[i+N*j] = 1.0f;
      }
}

void mprint(float *a, int N, int M)
{
   int i, j;

   for (int j = 0; j < M; j++)
   {
      for (int i = 0; i < M; i++)
         printf("%.2f ", a[i+N*j]);
      printf("...\n");
   }
   printf("...\n");
}
```

Matrix Representation in Memory

```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
        for (k = 0; k < n; ++k)
            a[i+n*j] += b[i+n*k] * c[k+n*j];
```

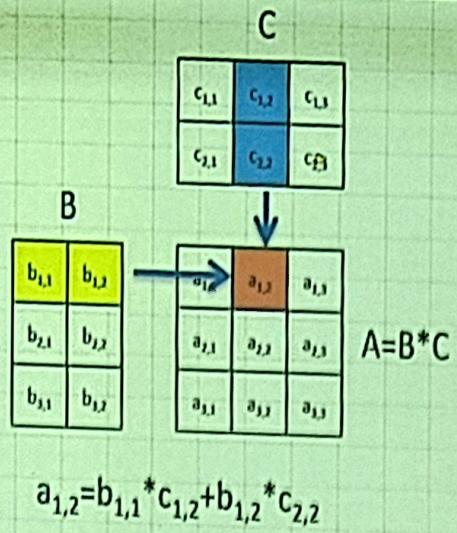
- Matrices are stored in column-major order



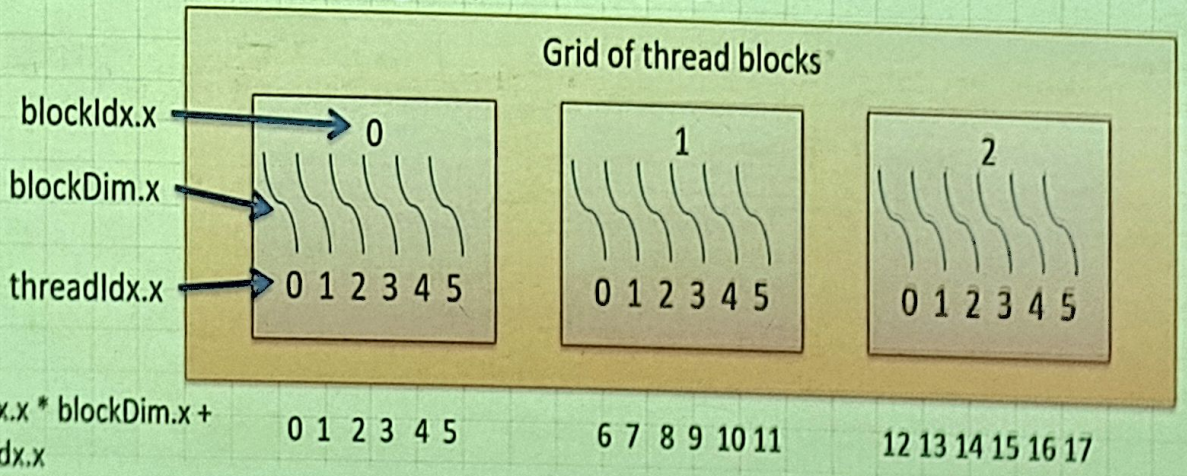
- For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

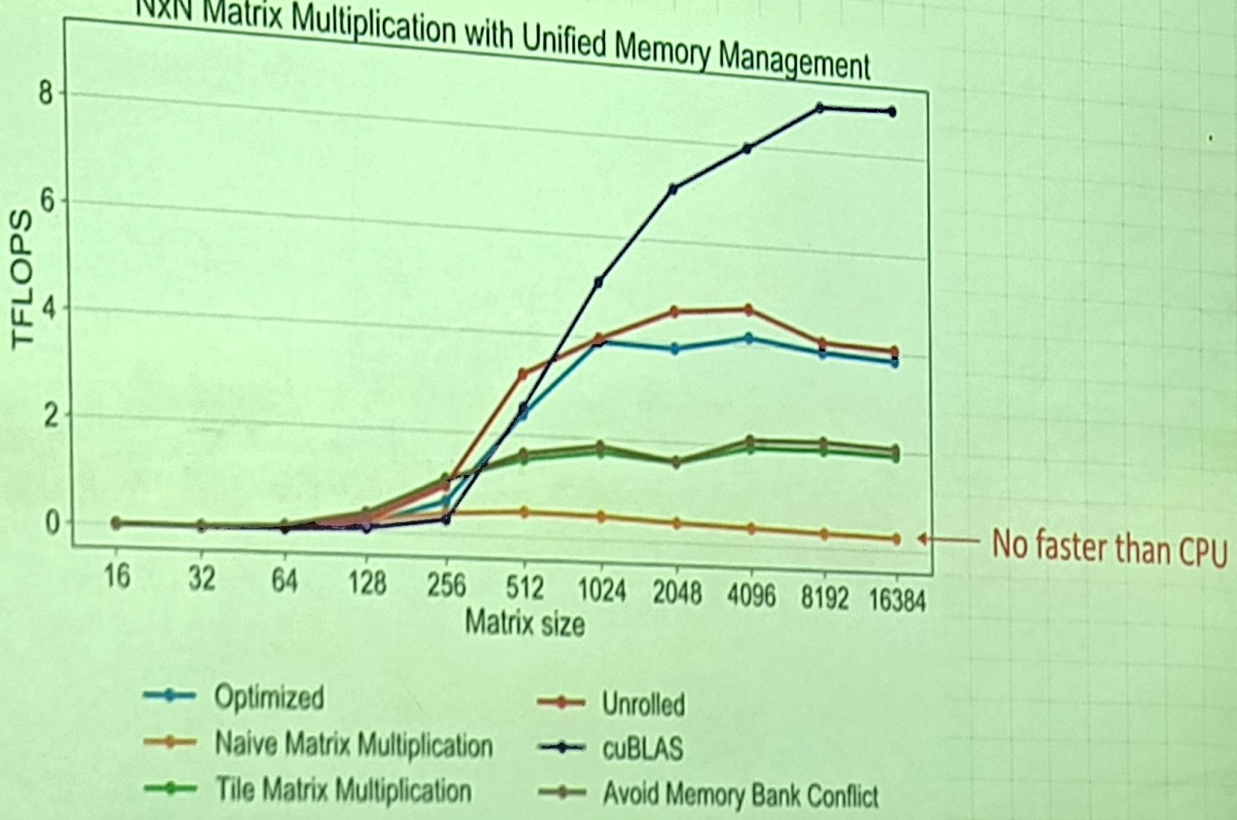
Map this code:

```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
        for (k = 0; k < n; ++k)
            a[i+n*j] += b[i+n*k] * c[k+n*j];
```



into this (logical) architecture:





Results from NVIDIA P100

et. al., "Efficient CUDA," 2017

Architecture knowledge is needed (again)

Kernel

```
void mmult(float *a, float *b, float *c, int N)
{
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         for (int k = 0; k < N; k++)
            a[i+j*N] += b[i+k*N]*c[k+j*N];
}
```

```
__global__
void mmult(float *a, float *b, float *c, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y;
   float sum = 0.0;

   for (int k = 0; k < N; k++)
      sum += b[i+N*k] * c[k+N*j];

   a[i+N*j] = sum;
}
```

```
dim3 dimGrid(32, 1024);
dim3 dimBlock(32);
mmult<<<dimGrid, dimBlock>>>(devPtrA, devPtrB, devPtrC, N);
```