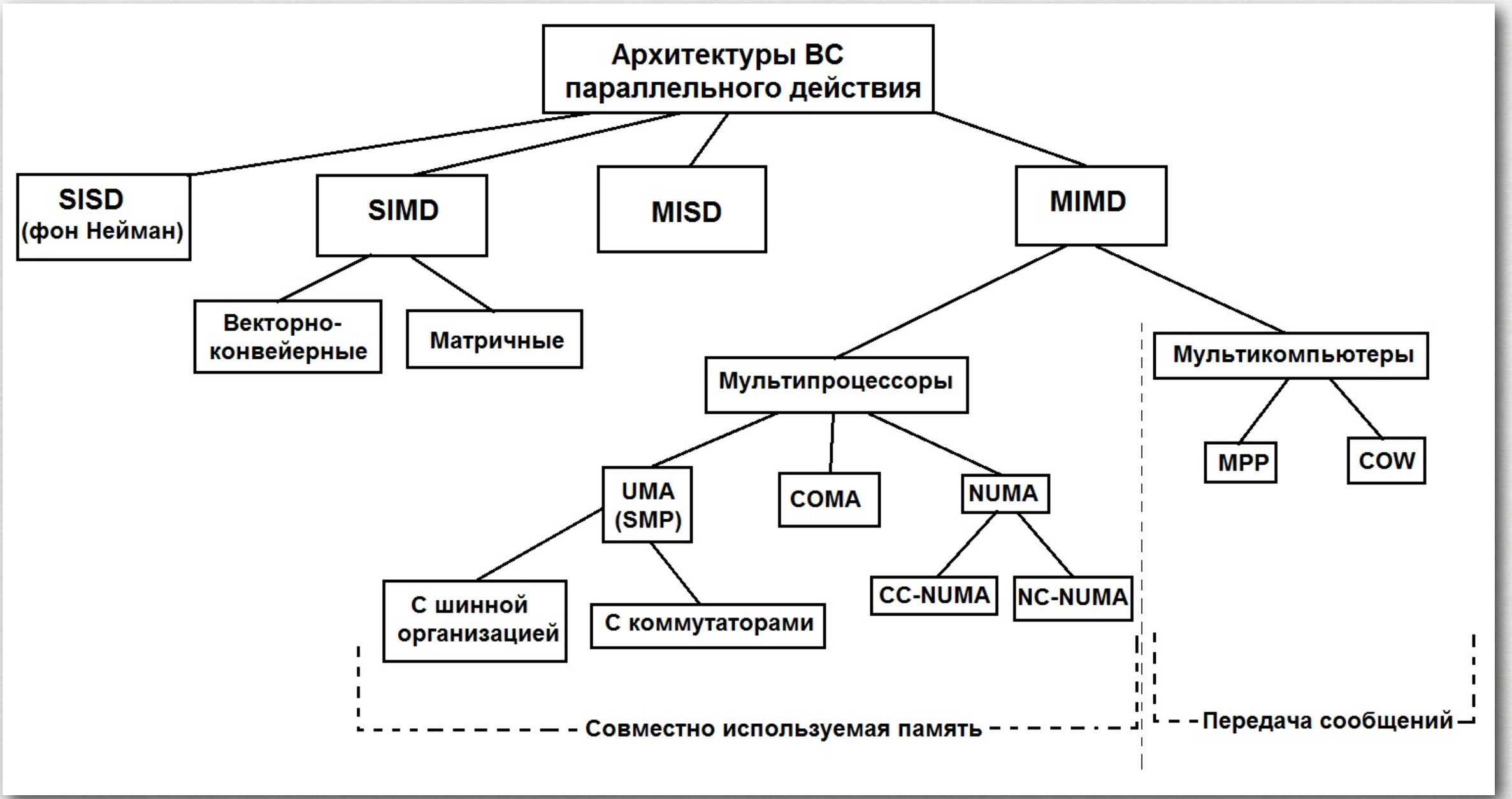
# ОСОБЕННОСТИ ВЫПОЛНЕНИЯ GPU-ПРОГРАММ. АСИНХРОННОСТЬ В CUDA, ОБРАБОТКА ОШИБОК, ИЗМЕРЕНИЕ ВРЕМЕНИ ВЫПОЛНЕНИЯ.

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

# МОДЕЛЬ ВЫПОЛНЕНИЯ CUDA-ПРОГРАММ

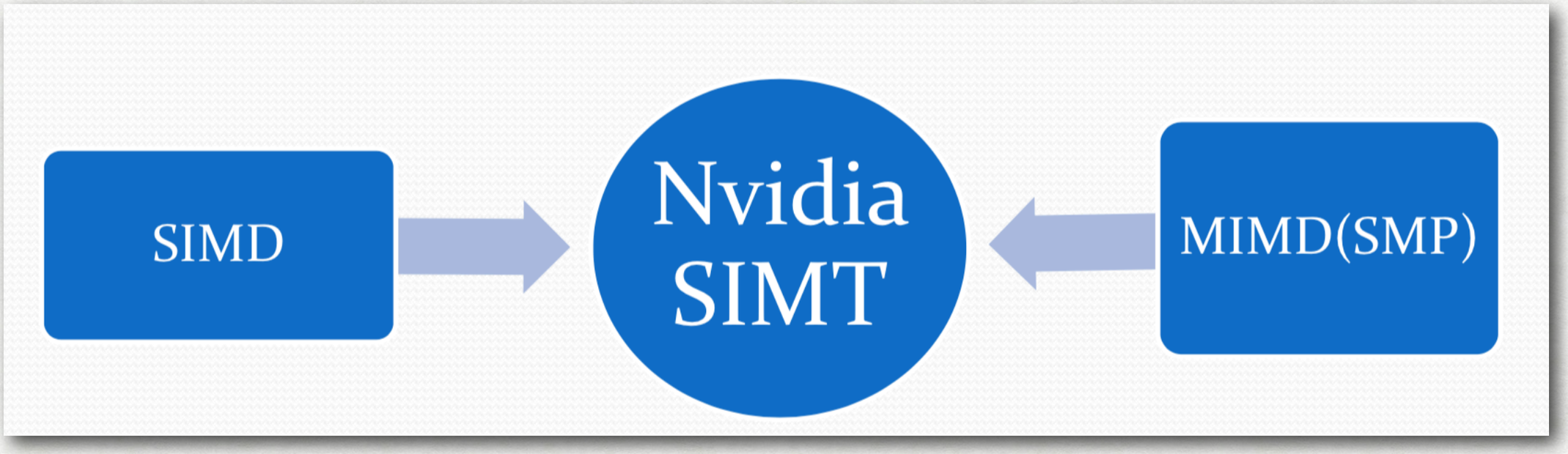
- На предыдущей лекции мы выяснили, что CUDA-ядра запускаются на множестве нитей
- Однако, необходимо понять, как CUDA-нити отображаются на архитектуру GPU (ядра, мультипроцессоры, кэши, память), а так же каким отличительными особенностями обладают
- Отличительные особенности:
    - SIMT модель вычислений (рассмотрена далее)
    - модель доступа к памяти нитей (рассмотрена на следующей лекции)
    - доступ нитей к иерархии памяти (более подробно во второй день)

# КЛАССИФИКАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ ПО ФЛИННУ



# А ГДЕ ЖЕ В ДАННОЙ КЛАССИФИКАЦИИ GPU?

- У NVIDIA собственная модель исполнения, имеющая черты как SIMD, так и MIMD
- Nvidia SIMT: Single Instruction – Multiple Thread



# SIMT: ВИРТУАЛЬНЫЕ НИТИ, БЛОКИ

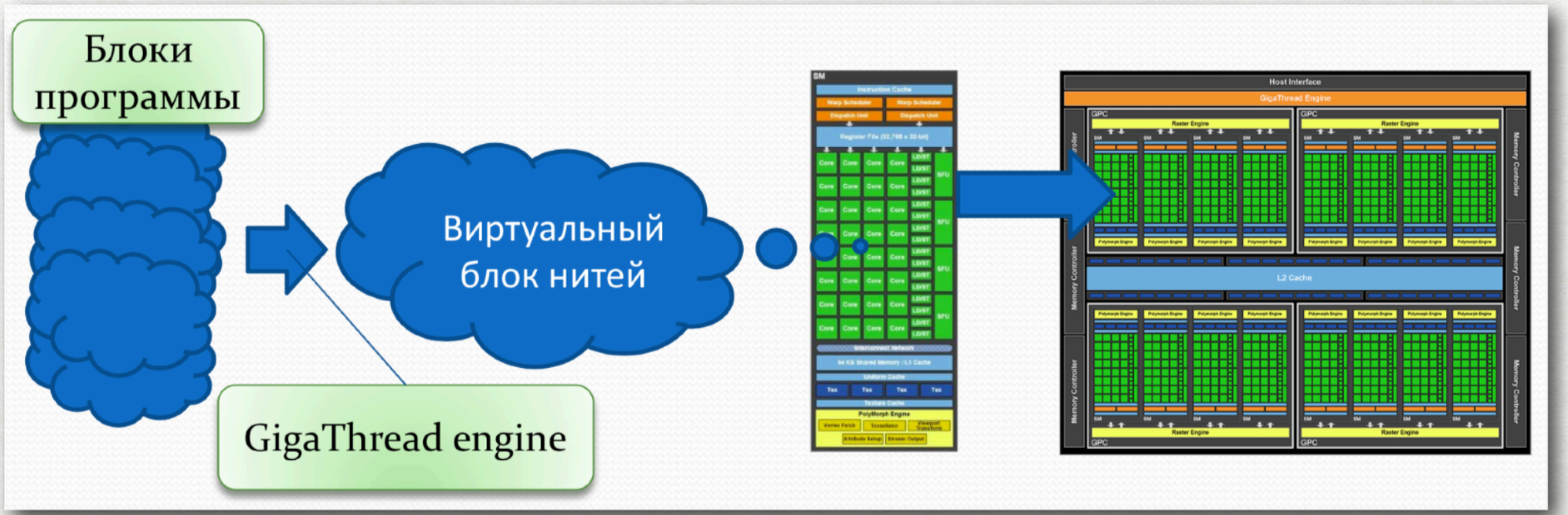
## (ВЫЧИСЛЕНИЯ С ТОЧКИ ЗРЕНИЯ ПРОГРАММИСТА)

- **Виртуально**, все нити
  - выполняются параллельно (MIMD)
  - Имеют одинаковые права на доступ к памяти (MIMD :SMP)
- Нити разделены на группы одинакового размера (блоки):
- В общем случае, **глобальная синхронизация всех нитей невозможна**, нити из разных блоков выполняются полностью независимо
- Есть локальная синхронизация внутри блока, нити из одного блока могут взаимодействовать через специальную память
- Нити не мигрируют между блоками. Каждая нить находится в своём блоке с начала выполнения и до конца

# SIMT: АППАРАТНОЕ ВЫПОЛНЕНИЕ

## (ВЫЧИСЛЕНИЯ С ТОЧКИ ЗРЕНИЯ АППАРАТУРЫ)

- Все нити из одного блока выполняются на одном мультипроцессоре (SM)
- Максимальное число нитей в блоке – 1024 (зависит от архитектуры)
- Блоки не мигрируют между SM
- Распределение блоков по мультироцесссорам непредсказуемо
- Каждый SM работает независимо от других
- Блоков может быть значительно больше, чем мультипроцессоров GPU

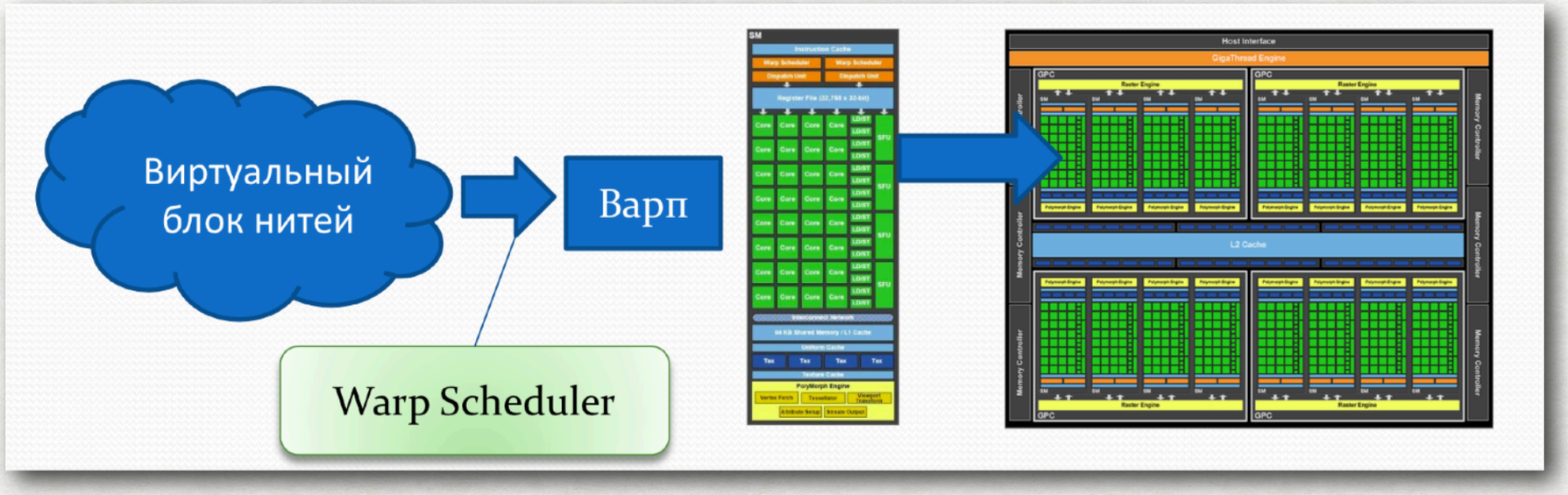


# ВАРПЫ (WARPS)

- Блоки нитей по фиксированному правилу (циклично в порядке увеличения индекса нити) разделяются на группы по 32 нити, называемые варпами (warp)

- Все нити варпа одновременно выполняют одну общую инструкцию (SIMD-выполнение)

- Warp Scheduler (планировщик варпов) на каждом цикле работы выбирает варп, все нити которого готовы к выполнению следующей инструкции и запускает весь варп



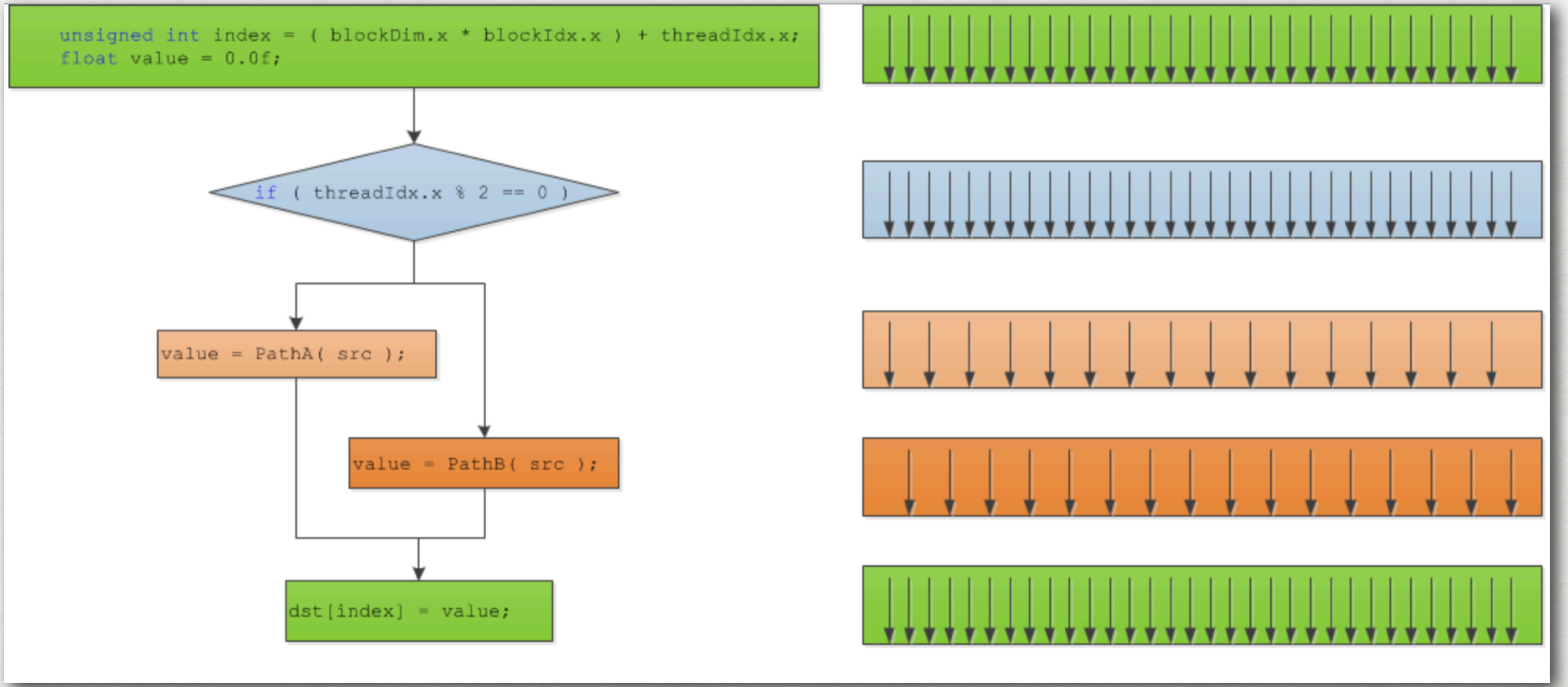
# ВЕТВЛЕНИЕ (BRANCHING)

- Если все нити варпа одновременно выполняют одну и ту же инструкцию, то как быть, если часть нитей эту инструкцию выполнять не должна?
- Пример:

if(<условие, зависшее от индекса нити>), где значение условия различается для нитей одного варпа

- Эти нити «замаскируются» нулями в специальном наборе регистров и не будут её выполнять, т.е. будут простаивать
- Другие примеры - switch+case, while, for ...

# ВЕТВЛЕНИЕ (BRANCHING)



# SIMD И ЗАГРУЗКА ДАННЫХ ИЗ ПАМЯТИ?

- Пусть warp выполняет инструкцию загрузки данных из памяти
- Аналогично с CPU, GPU использует кэши (рассмотрены на следующей лекции)
- Если хотя бы одна из нитей промахивается при попытке загрузить данные из кэша, все нити варпа будут ожидать загрузки данных

# АСИНХРОННОСТЬ В CUDA

- Чтобы GPU больше времени работало в фоновом режиме, параллельно с CPU, некоторые вызовы являются асинхронными (отправляют команду на устройство и сразу возвращают управление хосту)
- К таким вызовам относятся:
  - Запуски ядер
  - Копирование между двумя областями памяти на устройстве
  - Копирования, выполняемые функциями с окончанием *Async cudaMemSet – присваивает всем байтам области памяти на устройстве одинаковое значение (чаще всего используется для обнуления)

# АСИНХРОННОСТЬ В CUDA

- Почему тогда верно работает код?

*//запуск ядра (асинхронно)*

*sum_kernel<<< blocks, threads >>>(aDev, bDev, cDev);*

*//переслать результаты обратно на хост*

*cudaMemcpy(cHost, cDev, nb, cudaMemcpyDeviceToHost);*

- Ведь хост вызывает cudaMemcpy до завершения выполнения ядра!
- Ответ: Вызов ядра и cudaMemcpy попадают в один поток (поток по умолчанию), гарантируется их последовательное выполнение

# ОБРАБОТКА ОШИБОК

- Коды всех возникающих ошибок автоматически записываются в единственную специальную хостовую переменную типа enum cudaError_t
- Эта переменная в каждый момент времени равна коду последней ошибки, произошедшей в системе
  - cudaError_t cudaPeekAtLastError() – возвращает текущее значение этой переменной
  - cudaError_t cudaGetLastError() - возвращает текущее значение этой переменной и присваивает ей cudaSuccess
  - const char* cudaGetErrorString (cudaError_t error ) –по коду ошибки возвращает её текстовое описание

# ПРИМЕР ОБРАБОТКИ ОШИБОК

```
#define SAFE_CALL( CallInstruction ) { \
    cudaError_t cuerr = CallInstruction; \
    if(cuerr != cudaSuccess) { \
        printf("CUDA error: %s at call \"" #CallInstruction "\"\n", cudaGetErrorString(cuerr)); \
                throw "error in CUDA API function, aborting..."; \
    } \
}

#define SAFE_KERNEL_CALL( KernelCallInstruction ){ \
    KernelCallInstruction; \
    cudaError_t cuerr = cudaGetLastError(); \
    if(cuerr != cudaSuccess) { \
        printf("CUDA error in kernel launch: %s at kernel \"" #KernelCallInstruction "\"\n", cudaGetErrorString(cuerr)); \
                throw "error in CUDA kernel launch, aborting..."; \
    } \
    cuerr = cudaDeviceSynchronize(); \
    if(cuerr != cudaSuccess) { \
        printf("CUDA error in kernel execution: %s at kernel \"" #KernelCallInstruction "\"\n", cudaGetErrorString(cuerr));
\
                throw "error in CUDA kernel execution, aborting..."; \
    } \
}
```

# ЗАДАНИЯ

# ЗАДАНИЕ 1

## (SIMD МОДЕЛЬ ВЫЧИСЛЕНИЙ)

- добавить в сложение векторов ситуацию дивергенции нитей внутри варпа

  (1) if(threadIdx ... TODO)

  ```
      c[i] = a[i] + b[i];
  else
      c[i] = a[i] * b[i];
  ```

  (2) while(i < threadIdx)

  ```
  {
      //do somethin
      res_data[idx] += a[idx] * i;
  }
  ```

- Сравнить время выполнения с дивергенцией и без
- nvprof --metrics warp_execution_efficiency ./a.out

# ВОПРОСЫ?