ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ РАБОТЫ С ПАМЯТЬЮ. PINNED-ПАМЯТЬ. UNIFIED ПАМЯТЬ. АТОМАРНЫЕ ОПЕРАЦИИ.

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

ОПТИМИЗАЦИИ ПЕРЕСЫЛОК ДАННЫХ GPU ПРОГРАММ

- Время работы GPU приложения = Время выполнения ядер + **время копирования данных**
- Отложим на время в стороне вопрос оптимизации вычислительных ядер и обсудим, как можно ускорить время копирования данных
- 2 подхода:

1. **ускорение непосредственно копирований (Pinned memory)**
2. выполнение копирований параллельно с вычислениями (Cuda streams)

КАК ПОНЯТЬ, ЧТО НЕОБХОДИМО ОПТИМИЗИРОВАТЬ ПЕРЕДАЧИ ДАННЫХ?

снова помогает NVVP!

NVIDIA Visual Profiler

Analysis | *new_timeline | GPU Details (Summary) | Console | Settings

Export PDF Report

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.

Results

Low Memcpy/Kernel Overlap [0 ns / 28,303 ms = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low. Mor

Low Kernel Concurrency [0 ns / 128,309 s = 0%]
The percentage of time when two kernels are being executed in parallel is low. Mor

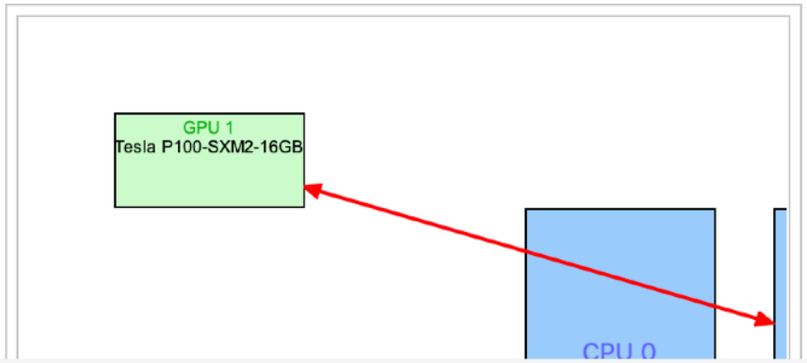
Low Memcpy Throughput [647.897 MB/s avg, for memcpys accounting for 17,2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. Mor

Low Memcpy Overlap [0 ns / 5,29 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low. Mor

NVLink Analysis

The following NVLink topology diagram shows logical NVLink connections between GPUs and CPUs. A logical NVLink can contain one or more physical links. When two devices A and B are connected by an NVLink, the receive throughput of device A is same as the transmit throughput of device B. The tables on right hand side show the properties for each logical NVLink.

* NVLink utilization may vary in accuracy, because any activity within the sampling period is treated as active, even though most of that period could be idle.



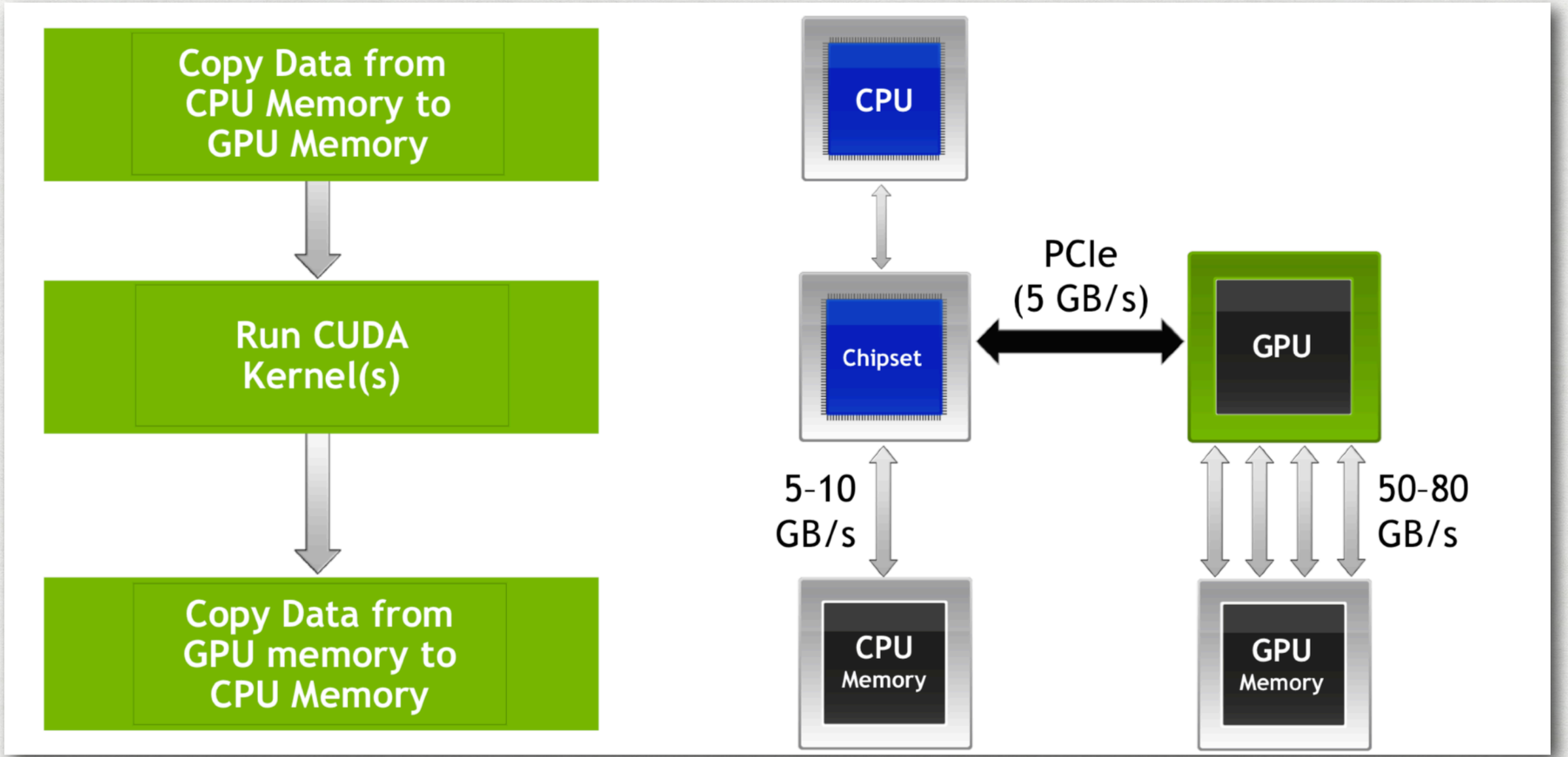
Logical NVLink Properties

Logical NVLink	Peak Bandwidth	Physical NVLinks	Peer Access	System Access	Peer Atomic	System Atomic	Utilization %	Idle time %
GPU0<-->...	80 GB/s	2	No	Yes	No	No	n/a	n/a
GPU1<-->...	80 GB/s	2	No	Yes	No	No	n/a	n/a

Logical NVLink Throughput

Logical NVLink	Avg Throughput	Max Throughput	Min Throughput
GPU0-->C...	n/a	n/a	n/a
GPU0<--C...	n/a	n/a	n/a
GPU1-->C...	n/a	n/a	n/a

КАК МЫ РАБОТАЛИ С КОПИРОВАНИЯМИ ДАННЫХ ДО ЭТОГО МОМЕНТА



- Вопрос: можно ли как-то ускорить копирования данных?

DMA AND ZERO-COPY

- Ответ - да, можно. Решение - использование DMA и Zero Copy.
- Zero-copy - копирование памяти без участия центрального процессора
- Копирование выполняется специальным контроллером, в то время как процессор переключается на другие задачи
- DMA (Direct memory access) – прямой доступ к оперативной памяти, без участия ЦП
- Реализуется через zero-copy операции
- Скорость передачи увеличивается, так как данные не пересылаются в ЦП и обратно

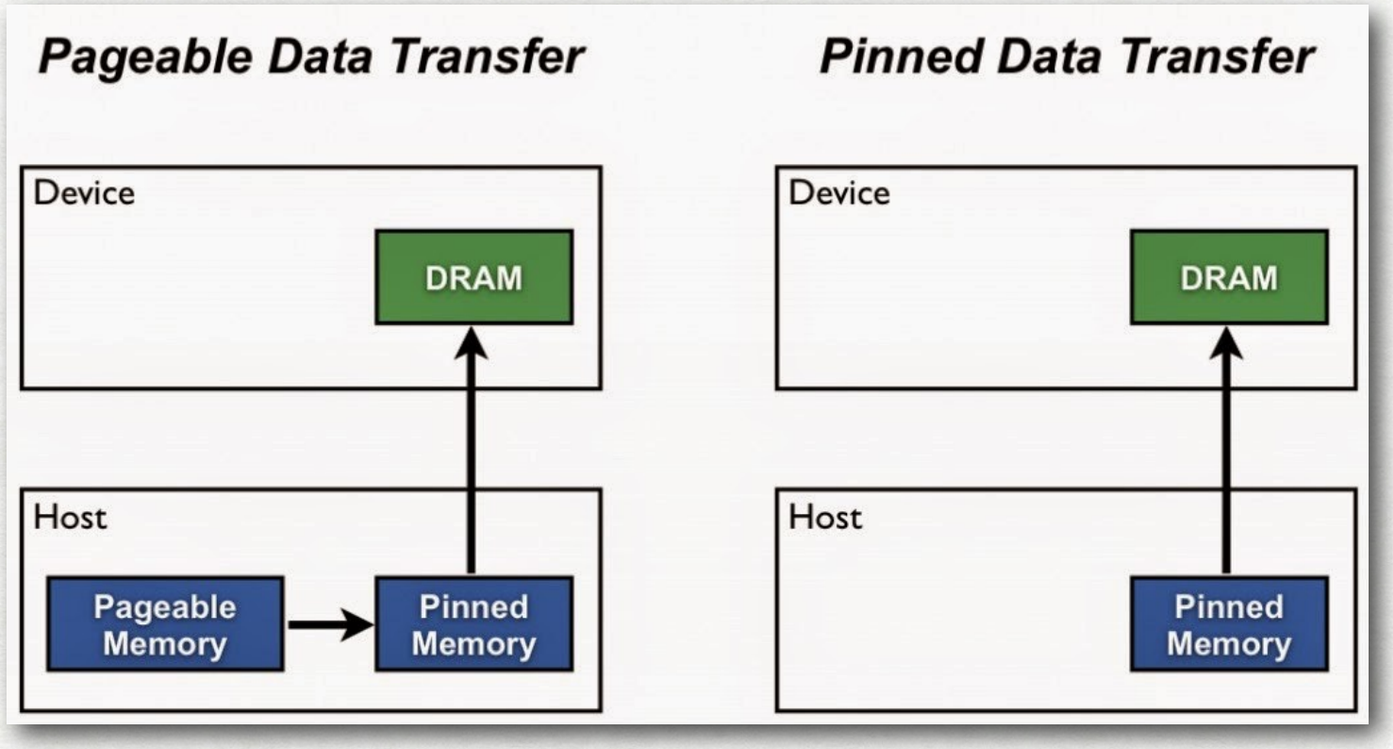
НО ЕСТЬ ПРОБЛЕМЫ...

- Выделение памяти на хосте (CPU) по умолчанию pageable
- Виртуальная память организованна в страницы, отображаемые на физические страницы ОП
- Виртуальные страницы могут быть перемещены в оперативной памяти или быть отгружены на диск
- В таких условиях реализовать DMA очень сложно

РЕШЕНИЕ ПРОБЛЕМЫ

- Запретим перемещение страниц по ОП и их выгрузку на диск
- Привяжем страницы виртуальной памяти к страницам физической
- Эти физические страницы теперь недоступны ОС для размещения новых виртуальных страниц (paging)
- page-able → page-locked
- Для page-locked памяти DMA реализуемо без существенных накладных расходов

PAGEABLE VS PINNED DATA TRANSFER



- GPU не может напрямую обращаться к pageable памяти, так что при выполнении копирования данных из pageable памяти в память устройства, CUDA-драйвер в начале должен выделить временную page-locked (pinned) память, скопировать в неё данные, а затем уже непосредственно выполнять передачу данных

CUDA & PINNED MEMORY

- В CUDA создать page-locked (pinned) память на хосте можно двумя способами:

  1. Сделать таковой выделенную ранее pageable память
  2. Выделить непосредственно pinned память

- Операции копирования с pinned памятью **происходят быстрее** и могут выполняться **параллельно с работой ядер**

ИСПОЛЬЗОВАНИЕ ФУНКЦИИ CUDAHOSTREGISTER

Сделать выделению обычным образом (malloc, new) память page-locked можно при помощи функции cudaHostRegister:

```
float *array = malloc(size * sizeof(float))

cudaHostRegister(array, size * sizeof(float),0);

...

cudaHostUnregister(ptr);
```

Копирование из pinned память выполняется аналогичным образом:

```
cudaMemcpy(device_array, array, size * sizeof(float), cudaMemcpyHostToDevice)
```

ПРЕИМУЩЕСТВА PINNED ПАМЯТИ

- Копирования обычной (pageable) памяти с хоста на GPU происходит через DMA к промежуточному буферу в pinned-памяти
- Поэтому копирование сразу из pinned-памяти быстрее – не нужно выделять память под буфер и копировать в него данные

ПОДДЕРЖИВАЕТ ЛИ ВАШЕ УСТРОЙСТВО PINNED-ПАМЯТЬ?

```
cudaDeviceProp prop;
 int whichDevice;
 checkCudaErrors( cudaGetDevice(&whichDevice));
 checkCudaErrors( cudaGetDeviceProperties(&prop, whichDevice));
 if (prop.canMapHostMemory != 1)
 {
    fprintf(stderr, "Device cannot map memory!\n»);
    return 1;
 }
checkCudaErrors(cudaSetDeviceFlags(cudaDeviceMapHost))
```

ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ CUDAMALLOCHOST/CUDAHOSTALLOC

Без флагов:

```
float *ptr = NULL;
cudaMallocHost(&ptr, n * sizeof(float));
```

Версия с флагами:

```
cudaHostAlloc(&ptr, n * sizeof(float), cudaHostAllocDefault);
```

BENCHMARKING PINNED TRANSFER

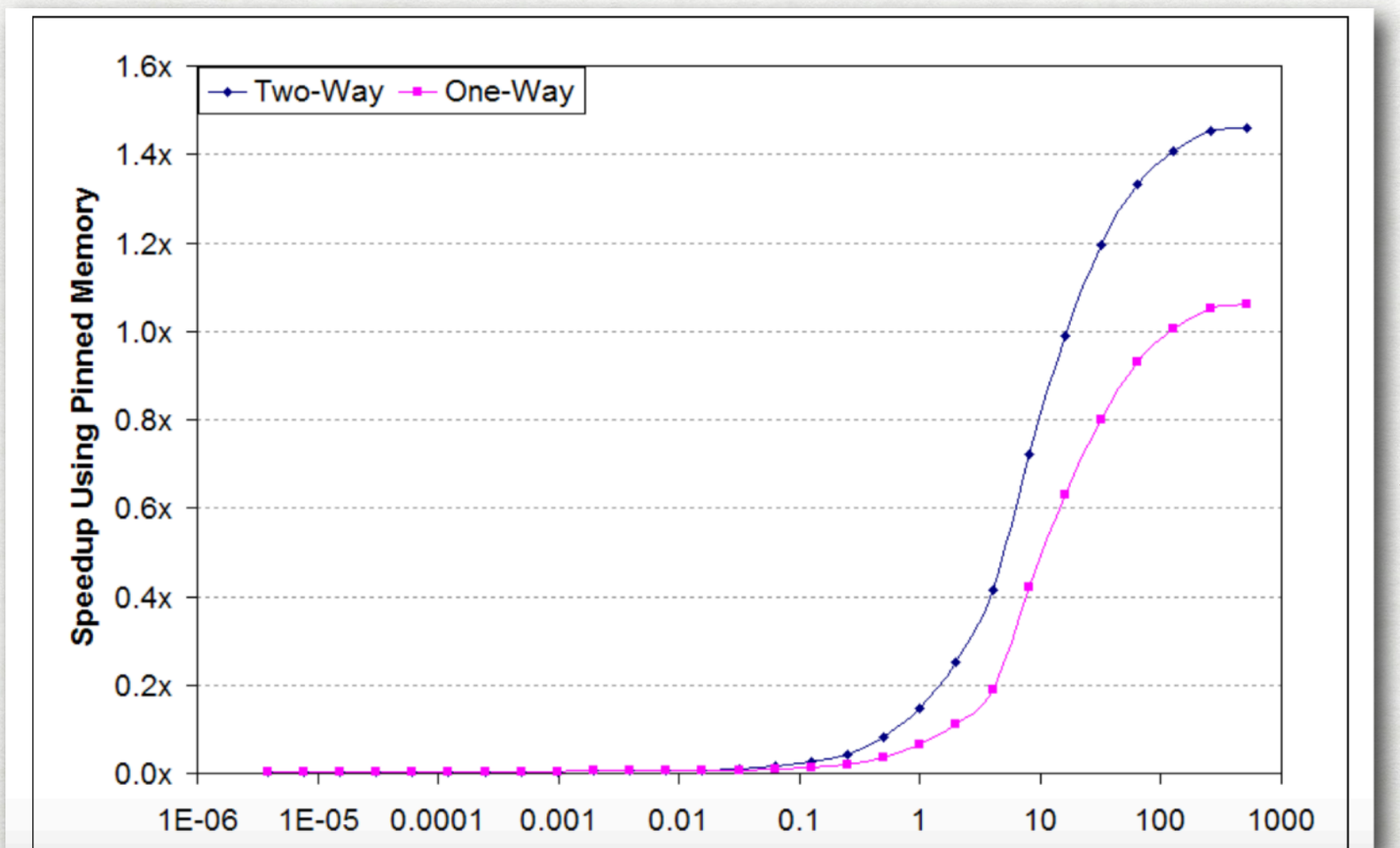
```
float *hostPtr = (float *)malloc(numberOfBytes);
cudaEventRecord(startPageable, 0);
// pagable transfer
cudaMemcpy(devicePtr, hostPtr, numberOfBytes, cudaMemcpyHostToDevice);
cudaEventRecord(stopPageable, 0);

cudaHostRegister(hostPtr, numberOfBytes, 0);
cudaEventRecord(startPinned, 0);

// pinned transfer
cudaMemcpy(devicePtr, hostPtr, numberOfBytes, cudaMemcpyHostToDevice);
cudaEventRecord(stopPinned, 0);
cudaDeviceSynchronize();
float elapsedPinned, elapsedPageable;
cudaEventElapsedTime(&elapsedPageable, startPageable, stopPageable);
cudaEventElapsedTime(&elapsedPinned, startPinned, stopPinned);
cout << "Pageable copy time: " << elapsedPageable << endl;
cout << "Pinned  copy time: " << elapsedPinned << endl;
```

в CUDA samples есть пример бенчмарка, можно его запустить!

```
Pageable copy time:  555.893066
Pinned  copy time: 339.375580
```

КОГДА ИСПОЛЬЗОВАТЬ PINNED ПАМЯТЬ?

- Выделение pinned памяти происходит дольше стандартного malloc
- Важно оценить, при каких размерах выделяемых данных выгодно использовать pinned память



Время на выделение, передачу и освобождение Pinned и non-pinned памяти.

ВЫВОДЫ

Плюсы:

- Операции копирования с pinned памятью происходят быстрее и могут выполняться параллельно с работой ядер
- Pinned память доступна как с хоста, так и с устройства

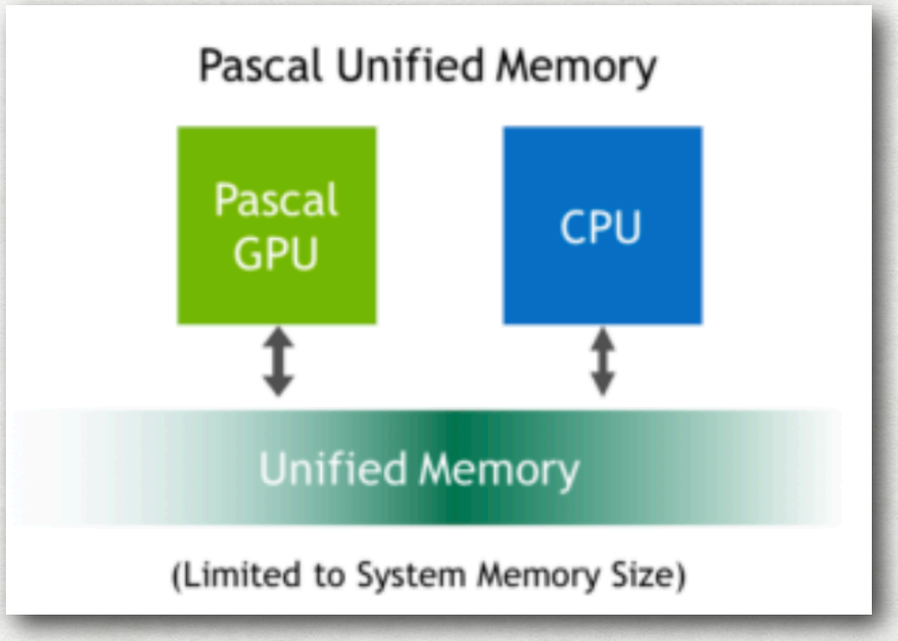
Минусы:

- Выделение pinned-памяти занимает больше времени, чем обычный malloc
- Доступный для выделения объем pinned памяти сильно ограничен
- Чрезмерное использование page-locked памяти может деградировать систему

UNIFIED ПАМЯТЬ

ЧТО ТАКОЕ UNIFIED-ПАМЯТЬ?

- Unified-память позволяет работать с единым адресным пространством, в которое входит **и память хоста, и память GPU**
- Т.е. можно:
  - выделить память на хосте (специальной функцией)
  - по полученному указателю произвести запись данных с хоста
  - вызвать ядро, передав ему этот же указатель для работы
  - а по его завершении - опять обратиться к данным со стороны хоста по тому же самому указателю
- При этом не требуется никакого явного копирования данных – всё **производится автоматически**



КАК РЕАЛИЗОВАНА UNIFIED ПАМЯТЬ?

- Для GPU поколений **до Pascal** перед запуском ядра производится полное копирование всех данных
- В архитектуре Pascal появился **Page Migration engine**
- Наличие Page Migration engine позволяет избежать ненужного копирования – при обращении GPU к памяти, которая еще не перенесена на данный GPU, происходит *page fault*.
- После чего производится копирование соответствующего участка памяти из CPU на GPU (page, страницы)

КАК РАБОТАТЬ С UNIFIED ПАМЯТЬЮ?

- Для выделения unified-память используется функция **cudaMallocManaged**

  cudaError_t cudaMallocManaged (void **ptr, size_t size, unsigned int flags = cudaMemAttachGlobal);

- Для освобождения – **cudaFree**
- При вызове **cudaMallocManaged** происходит выделение памяти на GPU, но возвращаемый указатель доступен как со стороны GPU (любого из установленных), так и со стороны CPU.
- Для GPU Pascal и выше при вызове **cudaMallocManaged** на самом деле может вообще не произойти никакого выделения реальной памяти – это произойдет при реальном обращении к памяти

ПРИМЕР ИСПОЛЬЗОВАНИЯ UNIFIED ПАМЯТИ

```
__global__ void kernel ( int * data, int n )
{
    size_t index = threadIdx.x + blockIdx.x * blockDim.x;

    if ( index < n )
        data [index] = index;
}

int main ()
{
    int   * ptr;
    int n = 1024*1024;      // # of elements
    int blockSize = 1024;
    int numBlocks = (n + blockSize - 1) / blockSize;
                    // allocate managed memory
    cudaMallocManaged ( &ptr, n * sizeof ( int ) );

    kernel<<<numBlocks, blockSize>>> ( ptr, n );

    cudaDeviceSynchronize ();

    for ( int i = 0; i < n; i++ )
        if ( ptr [i] != i )
        {
            printf ( "Error at %d - %d\n", i, ptr [i] );
            return 1;
        }

    cudaFree ( ptr );

    printf ( "Success\n" );

    return 0;
}
```

ПРЕИМУЩЕСТВА UNIFIED ПАМЯТИ

- Обычно объем памяти GPU ограничен (до 16-32 GB) и поэтому выделить много памяти на GPU – невозможно
- Однако поскольку при выделении через **cudaMallocManaged** мы фактически имеем дело с виртуальной памятью, то возможно выделять значительно большие объемы памяти
- Выделенная Unified память доступна как на хосте, так и **на всех** GPU системы
- При обращении к unified-памяти со стороны GPU (любого из) будет происходить автоматический перенос данных в память данного GPU, освобождая программиста от необходимости явно копировать данные
- Через unified память можно значительно проще реализовать динамические структуры данных – массивы массивов, списки, и др.

НЕДОСТАТКИ UNIFIED ПАМЯТИ

- Использование Unified памяти сильно упрощает жизнь программиста, но также оно обладает определенной ценой
- В результате быстродействие программы обычно оказывается ниже, чем при явном копировании данных
- Быстродействие программы так же становится менее предсказуемым и объяснимым – когда происходит выделение памяти? насколько часто копируются страницы? Подгружаются ли лишние данные с хоста?

ОПТИМИЗАЦИЯ ПРОГРАММ С UNIFIED ПАМЯТЬЮ

- Тем не менее, существую способы повысить скорость работы с unified-памятью. Для этого можно использовать функции **cudaMemAdvise** и **cudaMemPrefetchAsync**.
- **cudaMemAdvise** задает некоторые предпочтении для копирования памяти, что может повысить быстродействие.
- **cudaMemPrefetchAsync** задает ранее копирование (prefetch) данных еще до того, как произошло реальное обращение к ним.

Использование **cudaMemAdvise**

- При помощи функции **cudaMemAdvise** можно сообщать о предполагаемом характере использования памяти, что может помочь системе организовать работу более эффективно.
- cudaError_t cudaMemAdvise (const void * ptr, size_t size, cudaMemoryAdvice advice, int device);
- Параметр *advice* может принимать следующие значения:
  - cudaMemAdviseSetReadMostly - данное значение задает, что доступ к данной области памяти в основном будет на чтение;
  - cudaMemAdviseSetPreferredLocation - данное значение задает предпочтительное размещение данного блока в памяти устройства, задаваемого параметрами *device*;
  - cudaMemAdviseSetAccessedBy - задает, что обращения к данной области памяти будет идти со стороны устройства *device*;
  - и др.

Использование **cudaMemPrefetchAsync**

- cudaError_t cudaMemPrefetchAsync (const void * ptr, size_t size, int device, cudaStream_t stream = 0);
- служит для раннего копирования данных на заданное устройство (CPU или GPU).
- Параметр *ptr* задает адрес начала блока, который необходимо заранее скопировать в память, и параметр *size* задает размер блока в байтах.
- Задаваемый блок памяти должен целиком лежать в области памяти, выделенной через **cudaMallocManaged**.
- Параметр *device* задает устройство, в память которого необходимо произвести копирование.
- Если в качестве этого устройства выступает CPU, то этот параметр принимает значение **cudaCpuDeviceId**.

Пример использования **cudaMemPrefetchAsync**

```
int main ()
{
    int   * ptr;
    int n = 160*1024*1024;      // # of elements
    int blockSize = 1024;
    int numBlocks = (n + blockSize - 1) / blockSize;
    int device;

    cudaGetDevice ( &device );

    printf ( "Device %d\n", device );

                        // allocate managed memory
    cudaMallocManaged ( &ptr, n * sizeof ( int ) );

    for ( int i = 0; i < n; i++ )
        ptr [i] = i;

    cudaMemPrefetchAsync ( ptr, n * sizeof(int), device );

    kernel<<<numBlocks, blockSize>>> ( ptr, n );
```

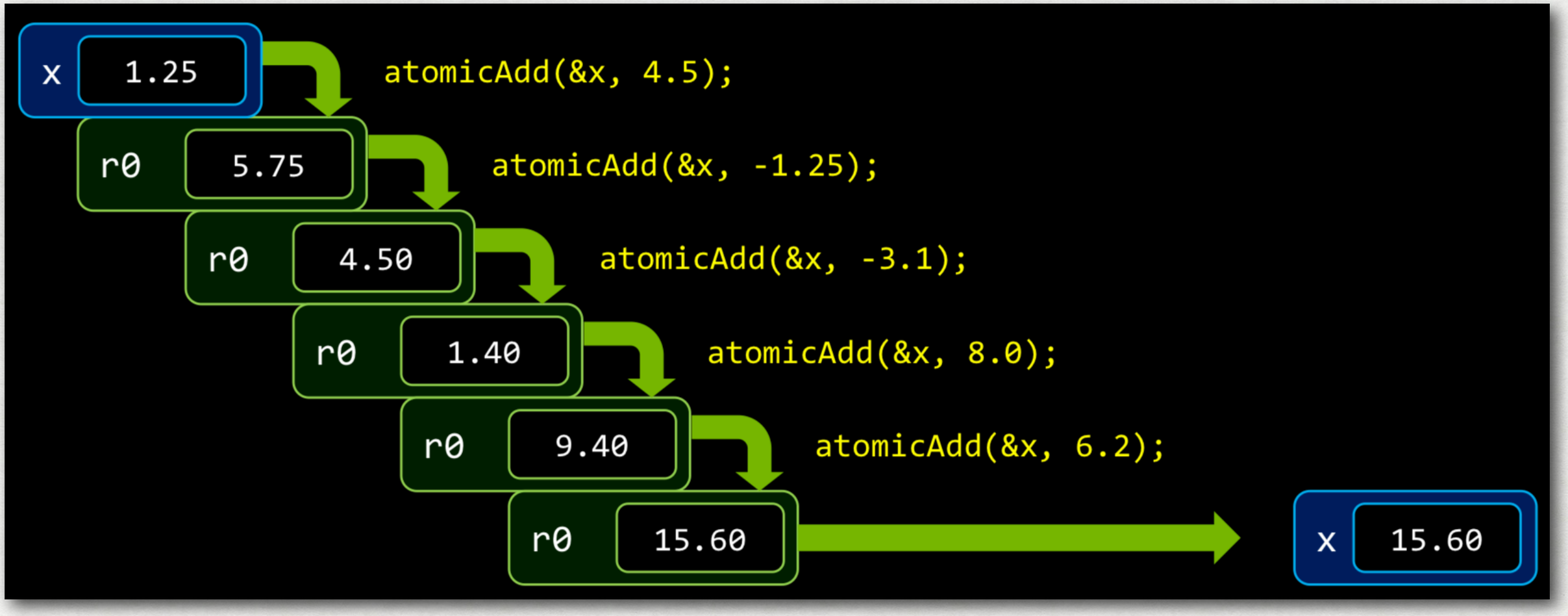
АТОМАРНЫЕ ФУНКЦИИ CUDA

- Атомарные операции в CUDA нужны для изменения race-conditions между потоками
- Атомарные операции в CUDA работают с global и shared memory
- Атомарные операции сериализуют обращения к памяти конкурирующих потоков, поэтому злоупотреблять ими не нужно
- Поддерживаются не все типы данных!
- `int atomicAdd(int* address, int val);`
- `int atomicSub(int* address, int val);`
- `int atomicAnd(int* address, int val);`
- `int atomicMax(int* address, int val);`
- `int atomicInc(unsigned int* address, unsigned int val);`
- `int atomicCAS(int* address, int compare, int val);` ((old == compare ? val : old)) -> memory

ЧТО ДЕЛАЕТ ATOMIC ADD?

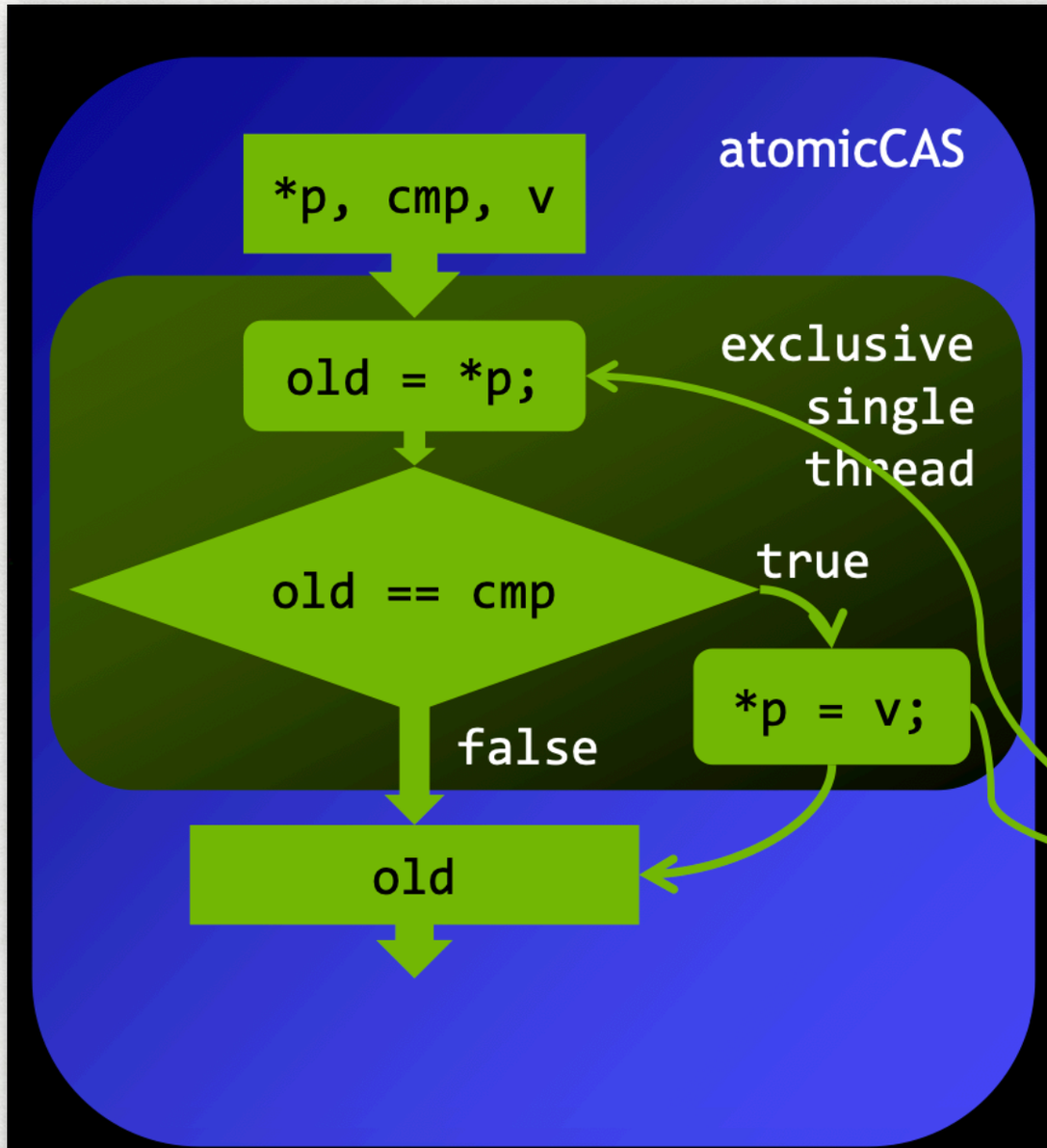
```
int atomicAdd(int *p, int v)
{
    int old;
    exclusive_single_thread
    {
        // atomically perform LD; ADD; ST ops
        old = *p; // Load from memory
        *p = old + v; // Store after adding v
    }
    return old;
}
```

ATOMIC ADD — ПРИМЕР

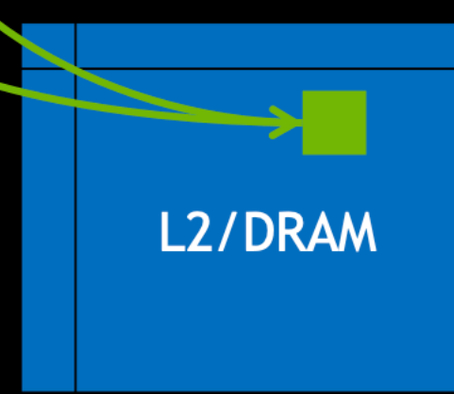


COMPARE AND SWAP

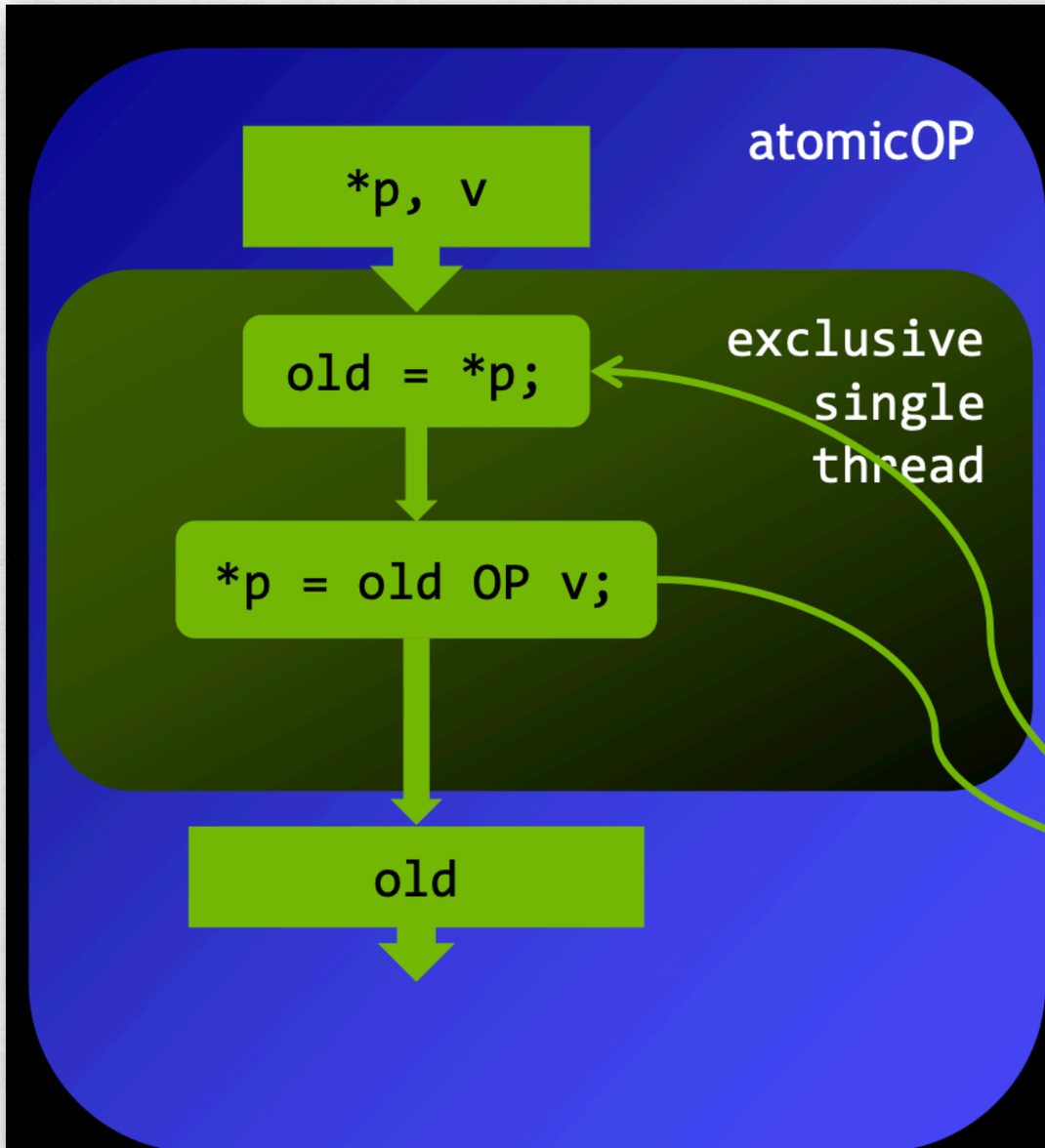
сравнивает значение в памяти с одним из аргументов, и в случае успеха записывает второй аргумент в память




```
int atomicCAS(int *p, int cmp, int v
{
    exclusive_single_thread
    {
        int old = *p;
        if (cmp == old) *p = v;
    }
    return old;
}
```



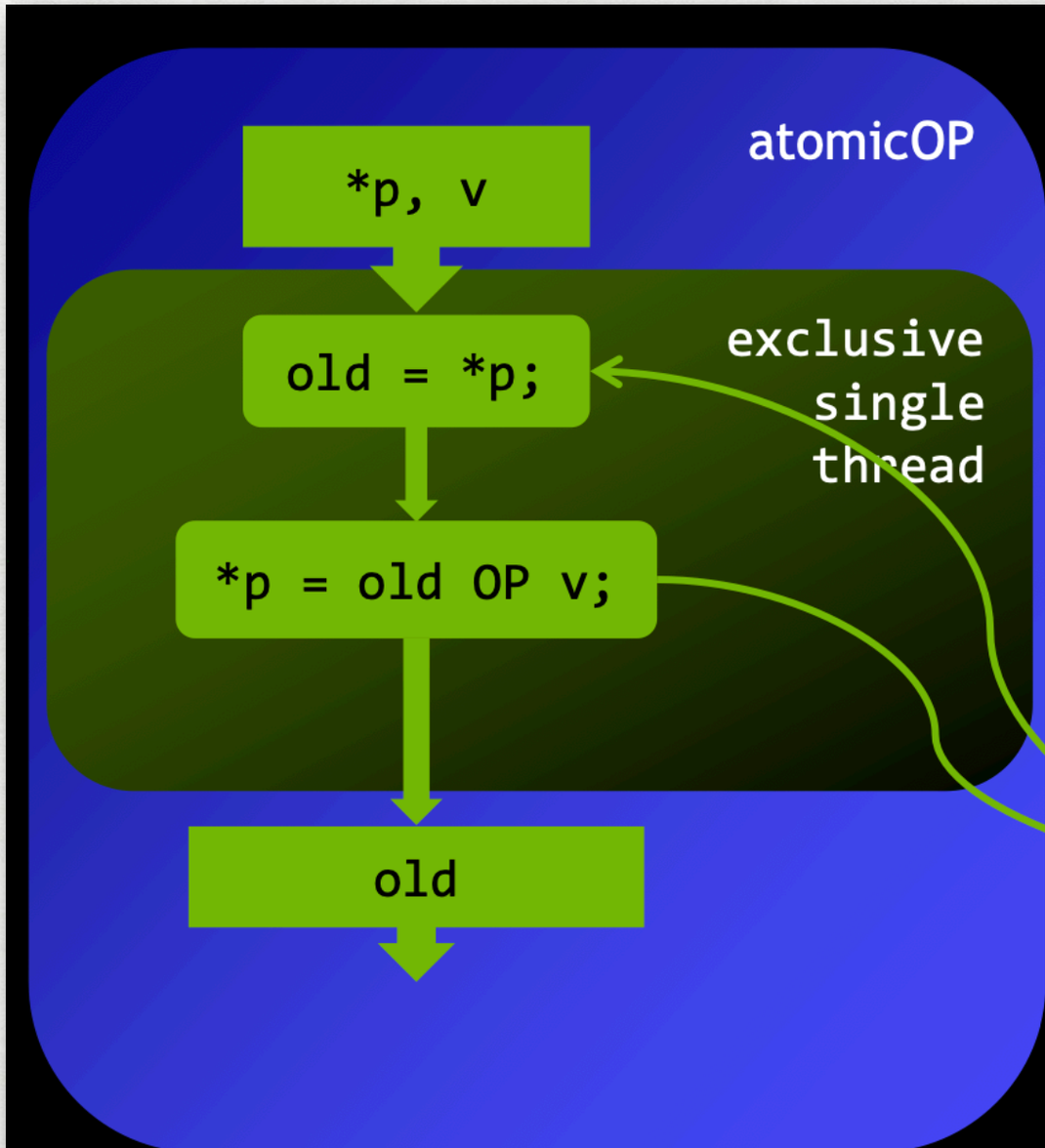
ГРУППА АРИФМЕТИКО-ЛОГИЧЕСКИХ АТОМАРНЫХ ОПЕРАЦИЙ




```
int atomicOP(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = old OP v;
    }
    return old;
}
```

Binary Ops:
Add, Min, Max
And, Or, Xor

ГРУППА АРИФМЕТИКО-ЛОГИЧЕСКИХ АТОМАРНЫХ ОПЕРАЦИЙ

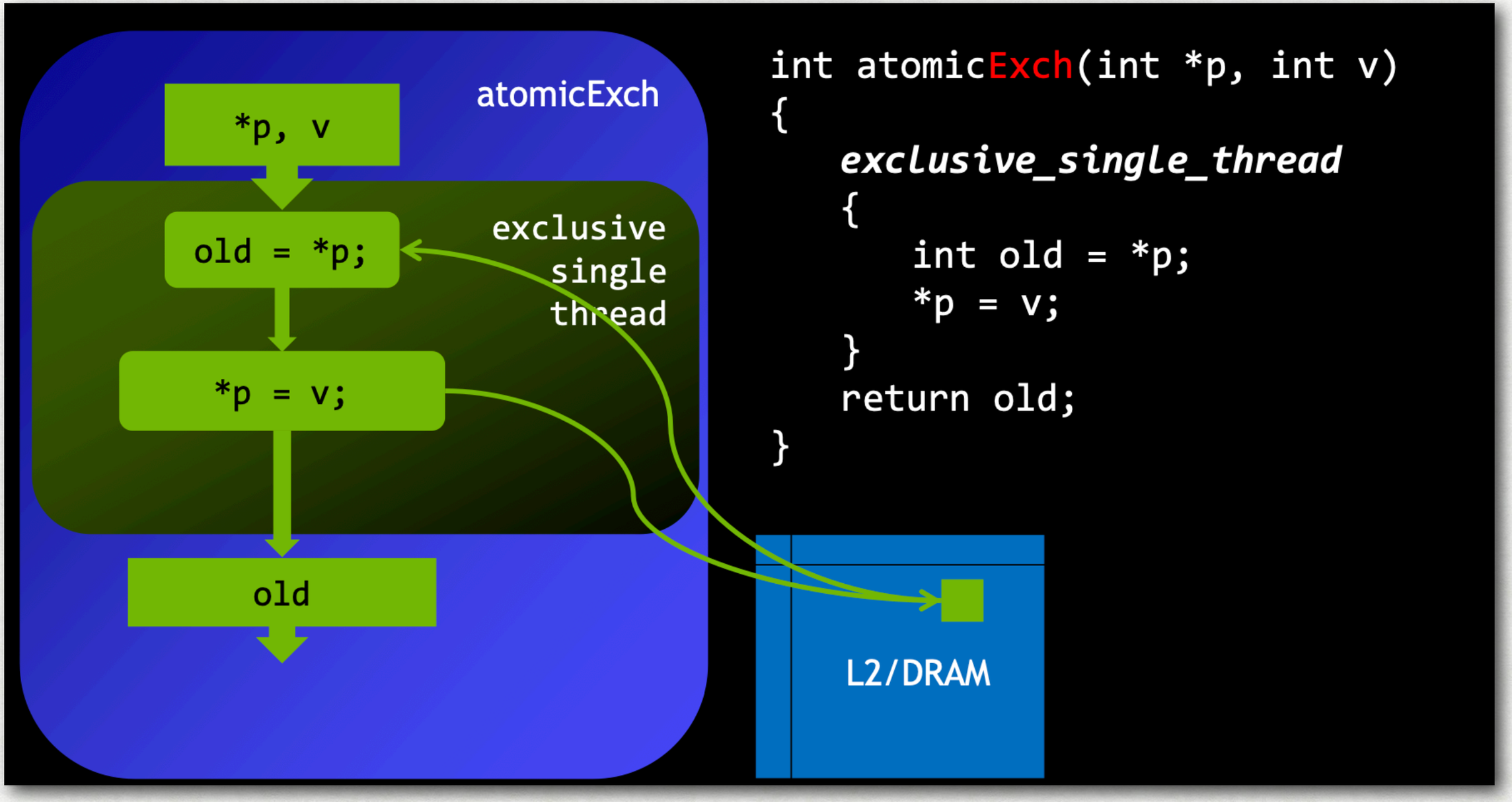


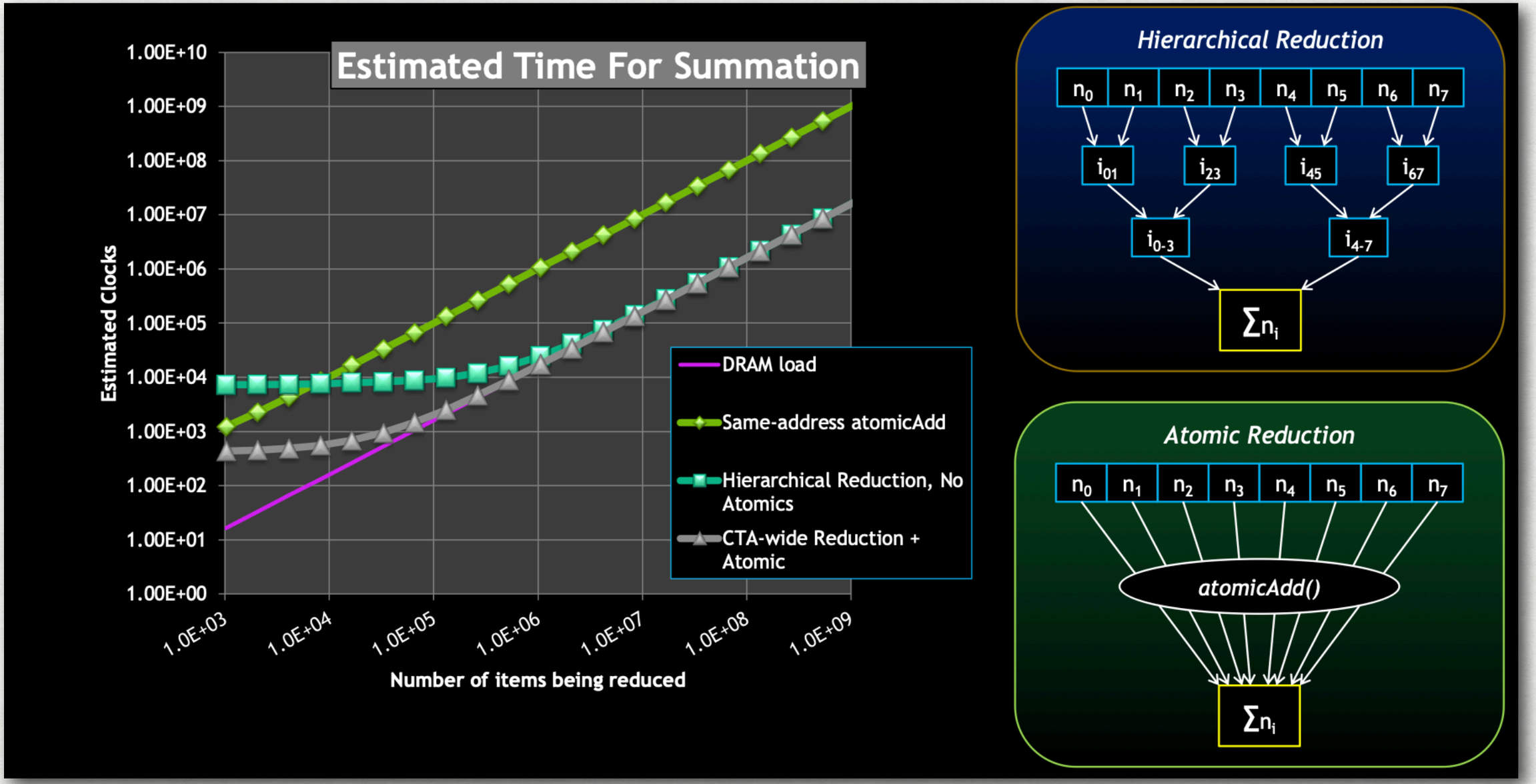

```
int atomicOP(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = old OP v;
    }
    return old;
}
```

Binary Ops:
Add, Min, Max
And, Or, Xor

ATOMIC-ОБМЕН

позволяет атомарно записать данные в память и получить значение, которое **было затёрто**



ATOMIC — РЕАЛИЗАЦИЯ РЕДУКЦИИ

```
for (int i = 0; i < N; i++)
        sum += A[i];
```