GPU TECHNOLOGY CONFERENCE

GPU-Accelerated Video Encoding

NVIDIA DevTech | Anton Obukhov <aobukhov@nvidia.com>

PRESENTED BY NVIDIA.

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Motivation for the talk

Video Encode and Decode are even more significant now as sharing of media is more popular with Portable Media Devices and on the internet

Encoding HD movies takes tens of hours on modern desktops

Portable and mobile devices have underutilized processing power

Trends

- GPU encoding is gaining wide adoption from developers (consumer, server, and professionals)
- Up to now the majority of encoders provide performance through a *Fast Profile*, but the *Quality Profile* increases CPU usage
- Normal state of things: CUDA exists since 2007
- A video encoder becomes mature after ~5 years
- Time to deliver quality GPU encoding

Trends

- Knuth: “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”
- x86 over-optimization has its roots in reference encoders
- Slices were a step towards partial encoder parallelization, but the whole idea is not GPU-friendly (doesn’t fit well with SIMT)
- Integrated solution is an encoder architecture from scratch

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Video encoding with NVIDIA GPU

Facilities:

- SW H.264 codec designed for CUDA
  - Baseline, Main, High profiles
  - CABAC, CAVLC

Interfaces:

- C library (NVCUVENC)
- Direct Show API

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Hybrid CPU-GPU encoding pipeline

- The codec should be designed with best practices for both CPU and GPU architectures

- PCI-e is the main bridge between architectures, it has limited bandwidth, CPU–GPU data transfers can take away all benefits of GPU speed-up

- PCI-e bandwidth is an order of magnitude less than video memory bandwidth

Hybrid CPU-GPU encoding pipeline

- There are not so many parts of a codec that vitally need data dependencies: CABAC, CAVLC are the most well-known

- Many other dependencies were introduced to respect CPU best practices guides, can be resolved by codec architecture revision

- It might be beneficial to perform some serial processing with one CUDA block and one CUDA thread on GPU instead of copying data back and forth

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

GPU Encoding Use Cases

- Video encoding
- Video transcoding
- Encoding live video input
- Compressing rendered scenes

GPU Encoding Use Cases

- The use cases differ in the way input video frames appear in GPU memory
- Frames come from GPU memory when:
  - Compressing rendered scenes
  - Transcoding using CUDA decoder
- Frames come from CPU memory when:
  - Encoding/Transcoding of a video file decoded on CPU
  - Live feed from a webcam or any other video input device via USB

GPU Encoding Use Cases

Helper libraries for source acquisition acceleration:

- NVCUVID – GPU acceleration of H.264, VC-1, MPEG2 decoding in hardware (available in CUDA C SDK)
- videoInput – an open-source library for video devices handling on CPU via DirectShow
  - DirectShow filter can be implemented instead to minimize the amount of “hidden” buffers on the CPU. The webcam filter writes frames directly into pinned/mapped CUDA memory

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Algorithm primitives

- Fundamental primitives of Parallel Programming
- Building blocks for other algorithms
- Have very efficient implementations
- Mostly well-known:
  - Reduce
  - Scan
  - Compact

Reduce

- Having a vector of numbers $a = [\ a_0, a_1, \ldots\ a_{n\text{-}1}\ ]$

- And a binary associative operator $\oplus$, calculate:

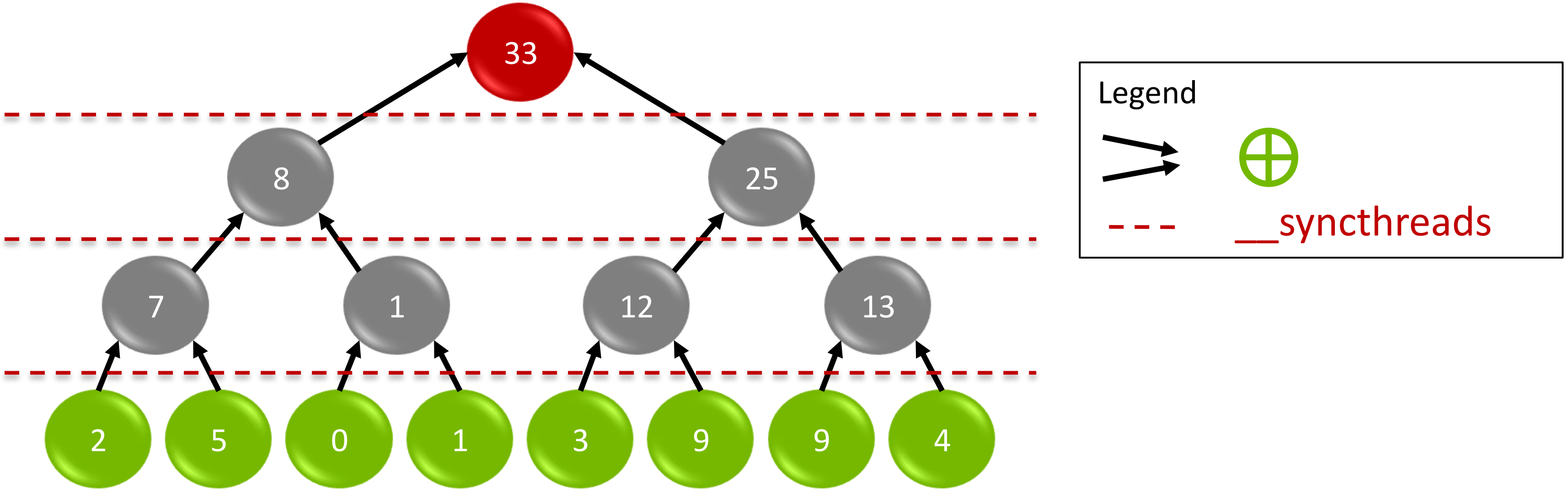
  $res = a_0 \oplus a_1 \oplus \ldots \oplus a_{n\text{-}1}$

- Instead of $\oplus$ take any of the following:

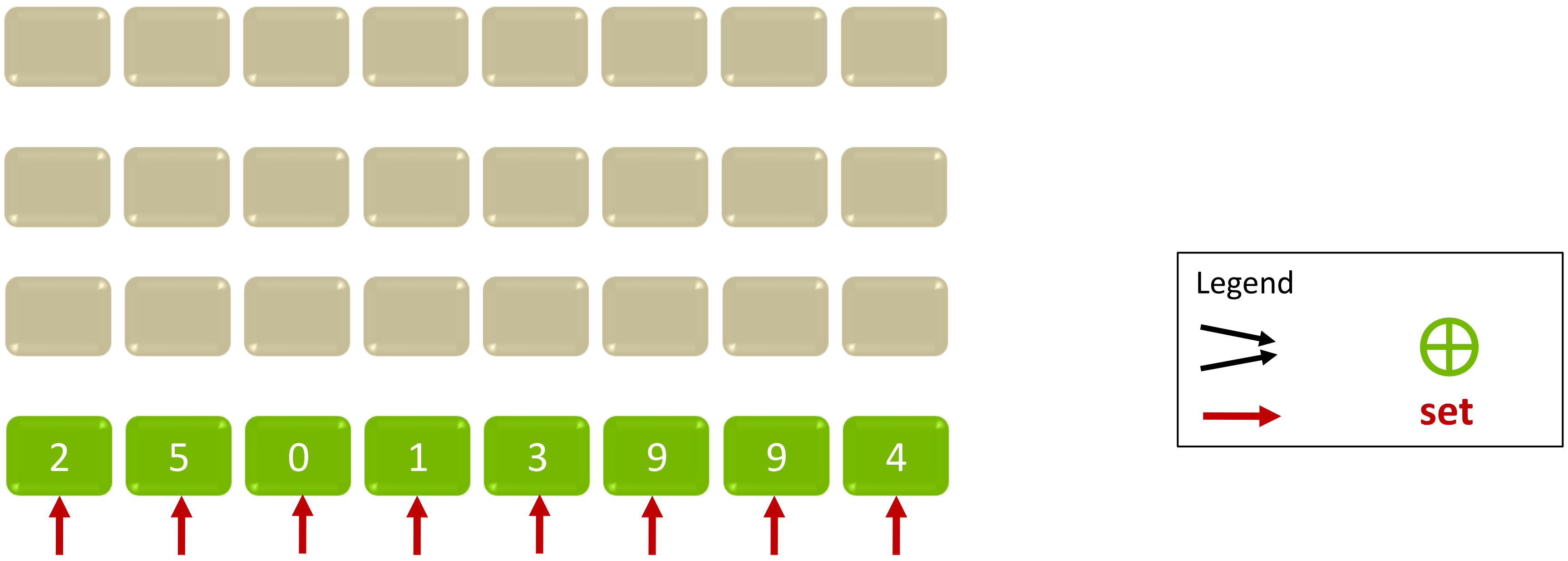
  "**+**", "*****", "**min**", "**max**", *etc*.

Reduce

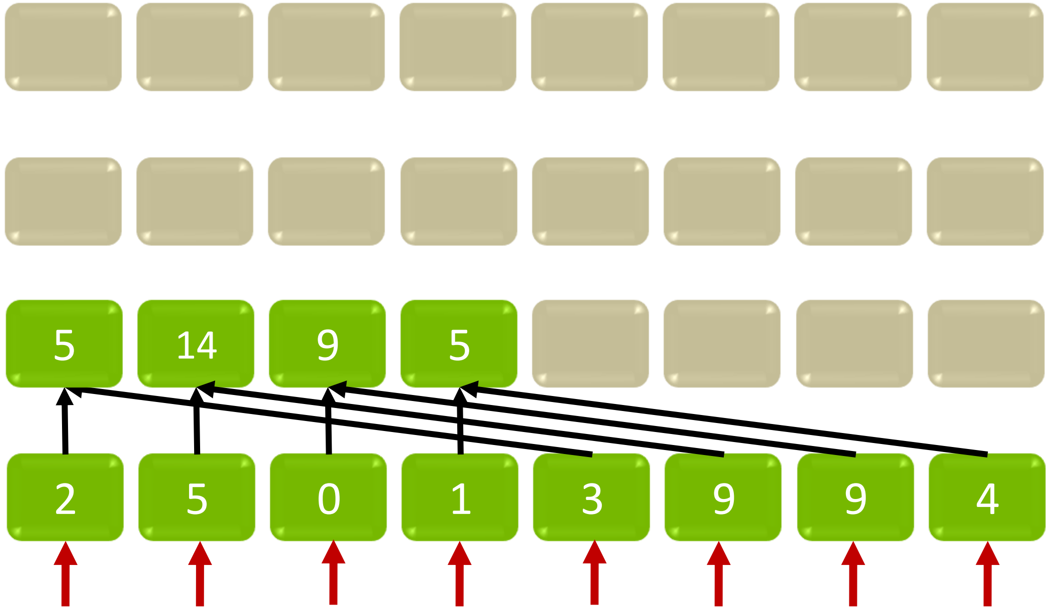
- Use half of N threads
- Each thread loads 2 elements, applies the operator to them and puts the intermediate result into the shared memory
- Repeat $\log_2 N$ times, halve the number of threads on every new iteration, use __syncthreads to verify integrity of results in shared memory

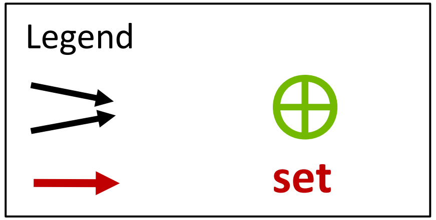


Reduce sample: 8 numbers

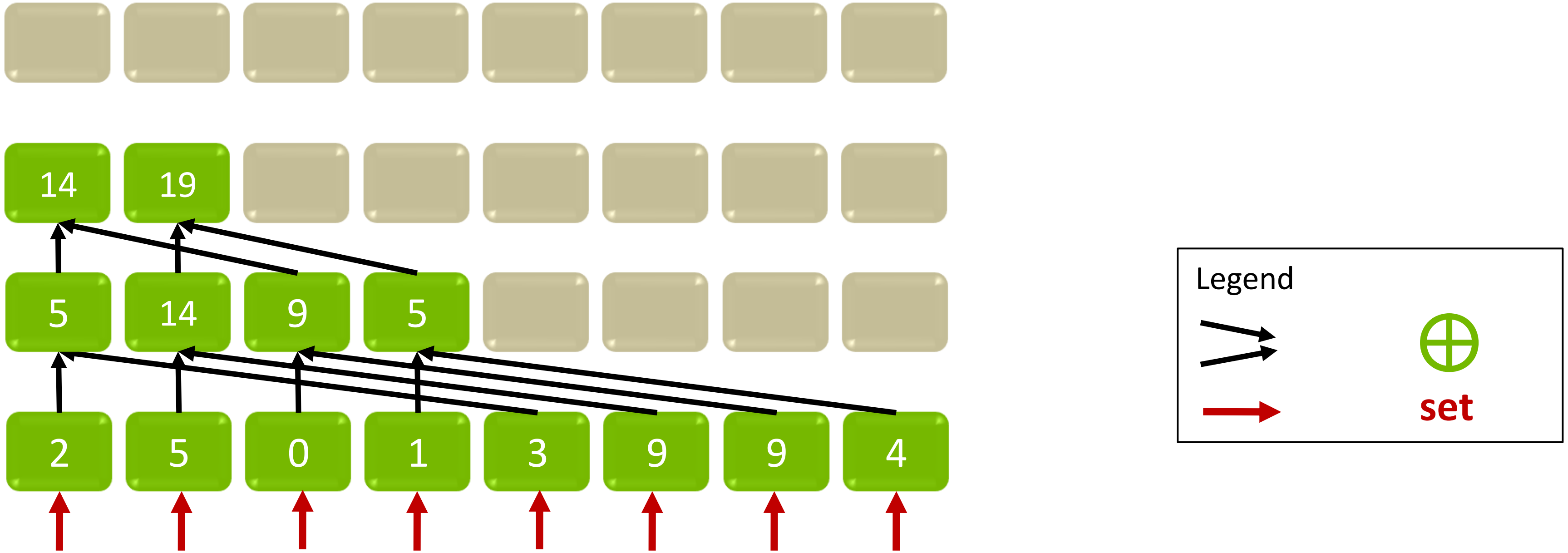


Reduce sample: 8 numbers

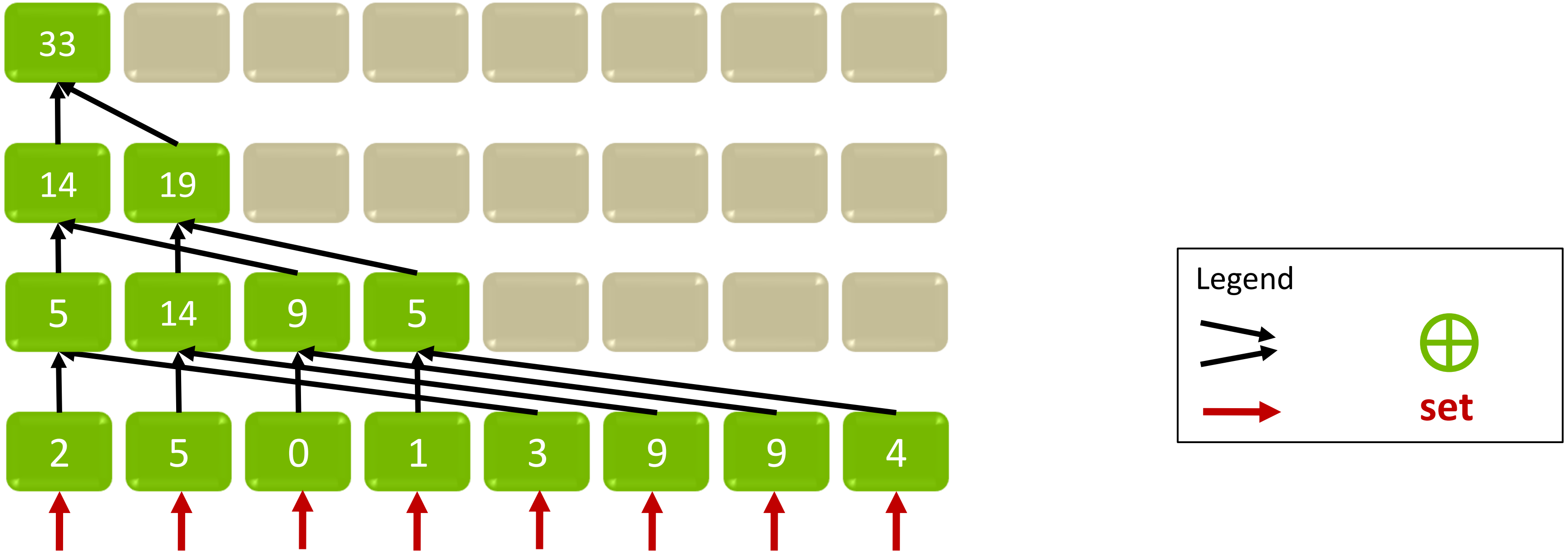




Reduce sample: 8 numbers



Reduce sample: 8 numbers



Scan

- Having a vector of numbers $a = [\ a_0, a_1, \ldots\ a_{n\text{-}1}\ ]$

- And a binary associative operator $\oplus$, calculate:

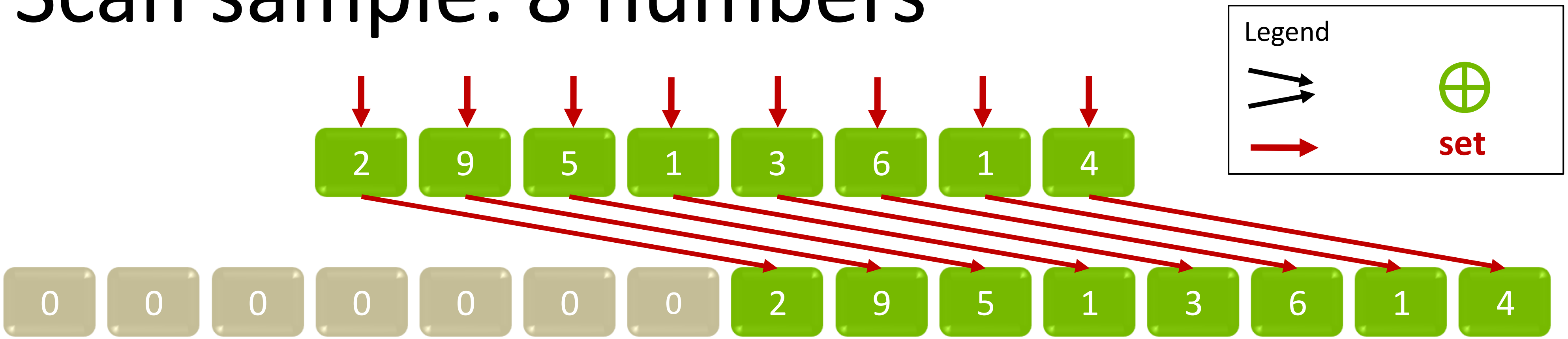
$$scan(a) = [\ 0, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n\text{-}2})\ ]$$

- Instead of $\oplus$ take any of the following:

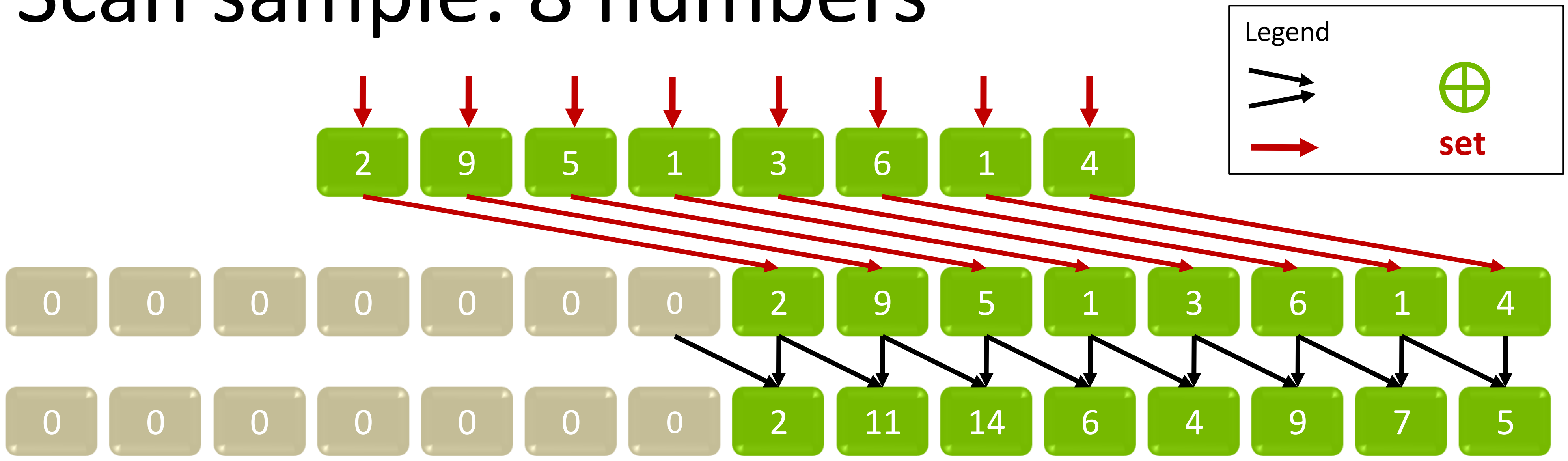
  " + ", " * ", "**min**", "**max**", *etc*.

Scan sample: 8 numbers

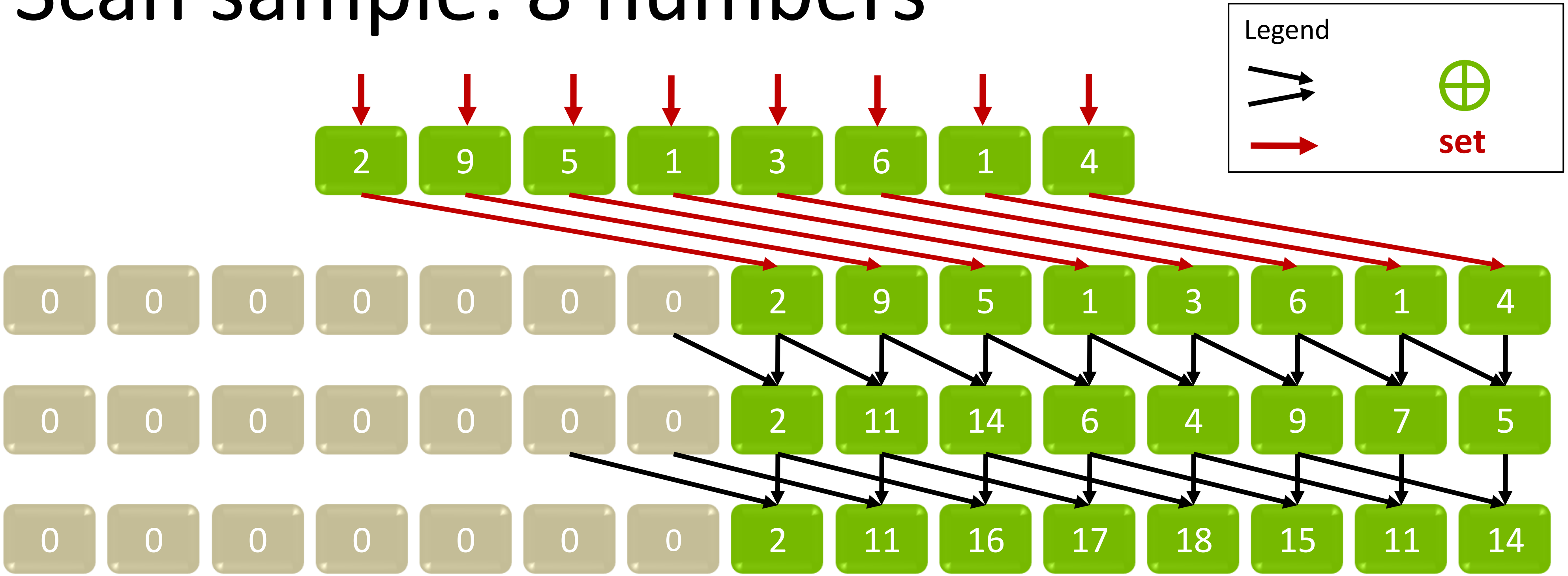
Scan sample: 8 numbers



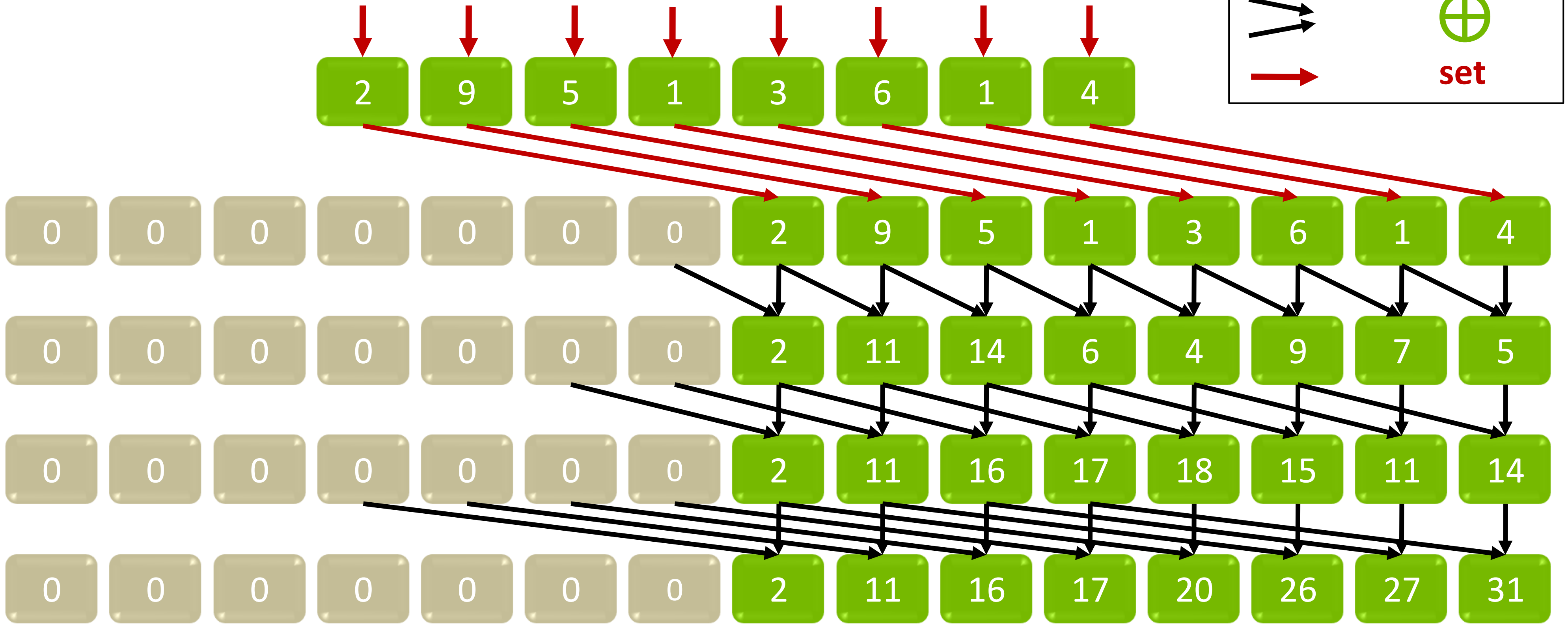
Scan sample: 8 numbers



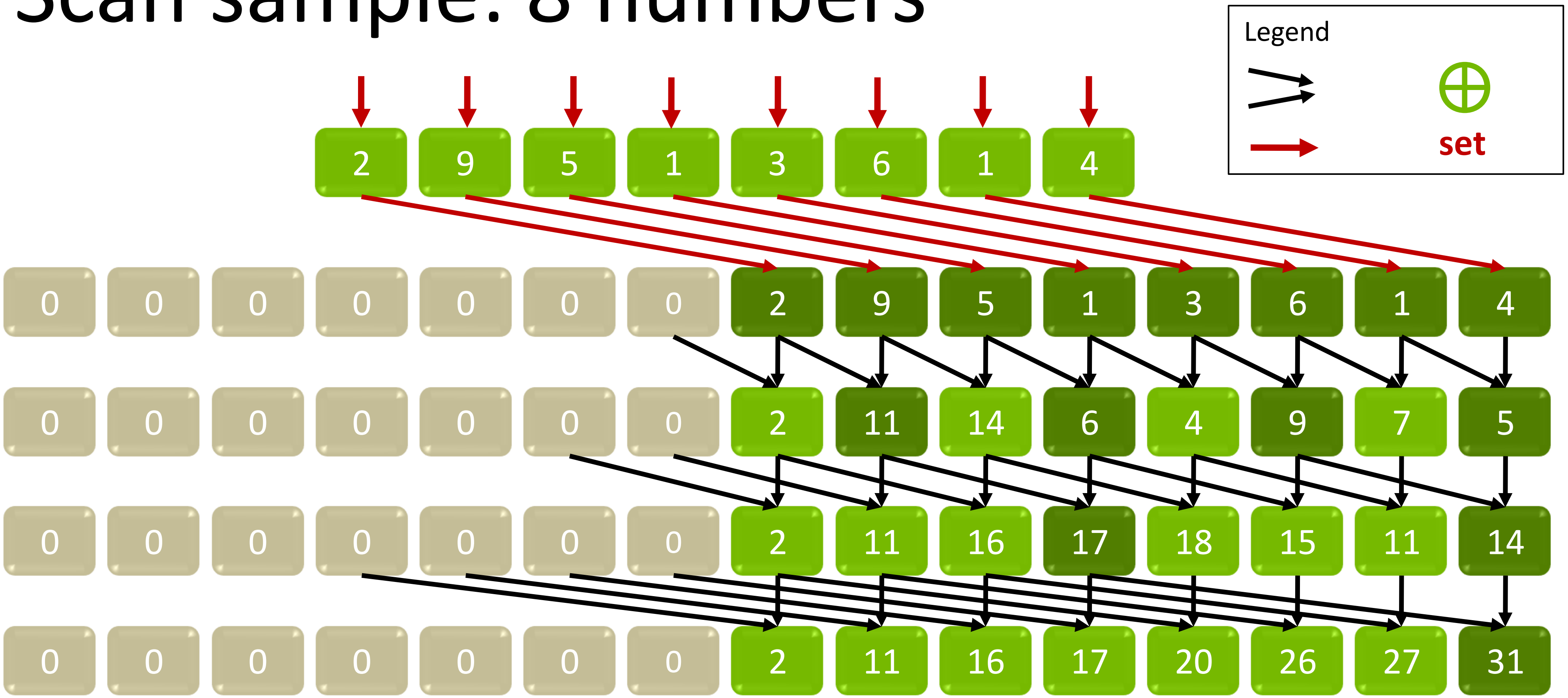
Scan sample: 8 numbers



Scan sample: 8 numbers



Scan sample: 8 numbers



Compact

- Having a vector of numbers $a = [\ a_0, a_1, \ldots\ a_{n\text{-}1}\ ]$ and a mask of elements of interest $m = [m_0, m_1, \ldots, m_{n\text{-}1}]$,

  $m_{\mathrm{i}} = [0 \mid 1]$,
- Calculate:

$$compact(a) = [\ a_{\mathrm{i}_0}, a_{\mathrm{i}_1}, \ldots\ a_{i_{\mathrm{k}\text{-}1}}], \text{ where}$$

$$reduce(m) = \mathrm{k},$$

$$\forall \mathrm{j} \in [0, \mathrm{k}\text{-}1]\colon m_{\mathrm{i}_\mathrm{j}} = 1.$$

Compact sample

Input:	0	1	2	3	4	5	6	7	8	9
Desired output:	0	2	4	7	8					

Green elements are of interest with corresponding mask elements set to 1, while gray to 0.

Compact sample

The algorithm makes use of the scan primitive:

Input:	0	1	2	3	4	5	6	7	8	9
Mask of interest:	1	0	1	0	1	0	0	1	1	0
Scan of the mask:	0	1	1	2	2	3	3	3	4	5
Compacted vector:	0	2	4	7	8					

Algorithm primitives discussion

- Highly efficient implementations in CUDA C SDK, CUDPP and Thrust libraries

- Performance benefits from HW native intrinsics (POPC and *etc*), which makes difference when implementing with CUDA

Algorithm primitives discussion

- Reduce: sum of absolute differences (SAD)

- Scan: Integral Image calculation, Compact facilitation

- Compact: Work pool index creation

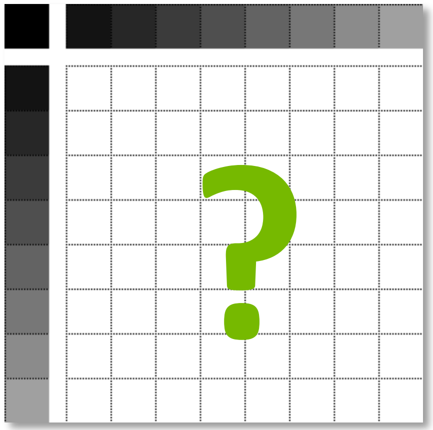
Outline

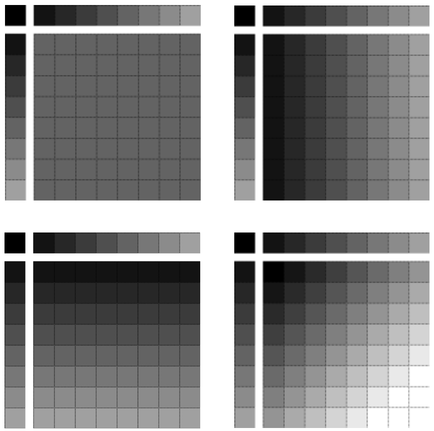
- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Intra Prediction

Intra Prediction

- Each block is predicted according to the predictors vector and the prediction rule
- Predictors vector consists of *N* pixels to the top of the block, *N* to the left and 1 pixel at top-left location
- Prediction rules (few basic for 16x16 blocks):
  1. **Constant** – each pixel of the block is predicted with a constant value
  2. **Vertical** – each row of the block is predicted using the top predictor
  3. **Horizontal** – each column of the block is predicted using the left predictor
  4. **Plane** – A 2D-plane prediction that optimally fits the source pixels





Intra Prediction

- Select the rule with the lowest prediction error

- The prediction error can be calculated as a Sum of Absolute Differences of the corresponding pixels in the predicted and the original blocks:

$$SAD = \sum_{x,y} \left| src[y][x] - predict[y][x] \right|$$

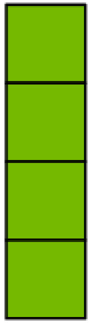
- Data compression effect: only need to store predictors and the prediction rule

Intra Prediction

Intra Prediction

CUDA: Intra Prediction 16x16

- An image block of 16x16 pixels is processed by 16 CUDA threads
- Each CUDA-block contains 64 threads and processes 4 image blocks at once:
  - 0-15
  - 16-31
  - 32-47
  - 48-63



- Block configuration: `blockDim = {16, 4, 1};`

CUDA: Intra Prediction 16x16

1. Load image block and predictors into the shared memory:

```
int mb_x = ( blockIdx.x * 16 ); // left offset (of the current 16x16 image block in the frame)
int mb_y = ( blockIdx.y * 64 ) + ( threadIdx.y * 16 ); // top offset
int thread_id = threadIdx.x; // thread ID [0,15] working inside of the current image block

// copy the entire image block into the shared memory
int offset = thread_id * 16; // offset of the thread_id row from the top of image block
byte* current = share_mem->current + offset; // pointer to the row in shared memory (SHMEM)
int pos = (mb_y + thread_id) * stride + mb_x; // position of the corresponding row in the frame
fetch16pixels(pos, current); // copy 16 pixels of the frame from GMEM to SHMEM

// copy predictors
pos = (mb_y - 1) * stride + mb_x; // position one pixel to the top of the current image block
if (!thread_id) // this condition is true only for one thread (zero) working on the image block
    fetch1pixel(pos - 1, share_mem->top_left); // copy 1-pix top-left predictor
fetch1pixel(pos + thread_id, share_mem->top[thread_id]); // copy thread_idth pixel of the top predictor
pos += stride - 1; // position one pixel to the left of the current image block
fetch1pixel(pos + thread_id * stride, share_mem->left[thread_id]); // copy left predictor
```

CUDA: Intra Prediction 16x16

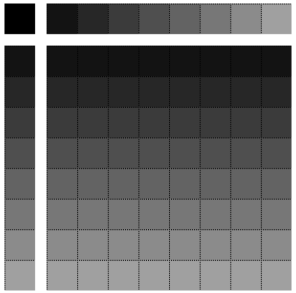
2. Calculate horizontal prediction (using the left predictor):

```
//INTRA_PRED_16x16_HORIZONTAL
int left = share_mem->left[thread_id]; // load thread_id pixel of the left predictor

int sad = 0;
for (int i=0; i<16; i++) // iterate pixels of the thread_idth row of the image block
{
    sad = __usad(left, current[i], sad); // accumulate row-wise SAD
}

// each of 16 threads own its own variable "sad", which resides in a register
// the function devSum16 sums all 16 variables using the reduce primitive and SHMEM
sad = devSum16(share_mem, sad, thread_id);

if (!thread_id) // executes once for an image block
{
    share_mem->mode[MODE_H].mode = INTRA_PRED_16x16_HORIZONTAL;
    // store sad if left predictor actually exists
    // otherwise store any number greater than the worst const score
    share_mem->mode[MODE_H].score = mb_x ? sad : WORST_INTRA_SCORE;
}
```



CUDA: Intra Prediction 16x16

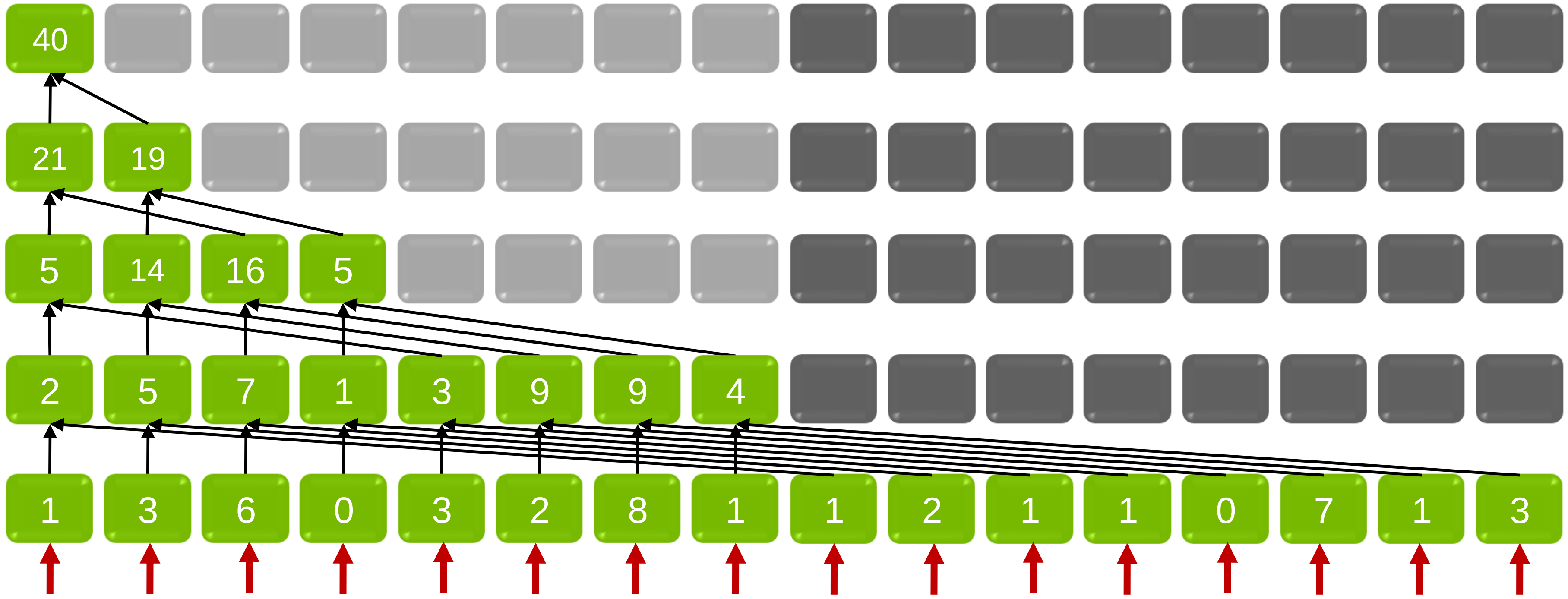
3.
```
__device__ int devSum16( IntraPred16x16FullSearchShareMem_t* share_mem, int a, int thread_id )
{
    share_mem->tmp[thread_id] = a; // store the register value in the shared memory vector
    //consisting of 16 cells
    __syncthreads(); // make all threads pass the barrier when working with shared memory

    if ( thread_id < 8 ) // work with only 8 first threads out of 16
    {
        share_mem->tmp[thread_id] += share_mem->tmp[thread_id + 8];
        __syncthreads();
        share_mem->tmp[thread_id] += share_mem->tmp[thread_id + 4]; // only 4 threads are useful
        __syncthreads();
        share_mem->tmp[thread_id] += share_mem->tmp[thread_id + 2]; // only 2 threads are useful
        __syncthreads();
        share_mem->tmp[thread_id] += share_mem->tmp[thread_id + 1]; // only 1 thread is useful
        __syncthreads();
    }
    return share_mem->tmp[0];
}
```

Under the certain assumptions about the CUDA warp size, some of the `__syncthreads` can be omitted

CUDA: Intra Prediction 16x16

3. `__device__ int devSum16`



CUDA: Intra Prediction 16x16

4. Calculate rest of the rules (Vertical and Plane)
5. Rule selection: take the one with the minimum SAD
6. Possible to do Intra reconstruction in the same kernel (see gray images few slides back)

CUDA Host: Intra Prediction 16x16

```
// main() body
byte *h_frame, *d_frame; // frame pointers in CPU (host) and CUDA (device) memory
int *h_ip16modes, *d_ip16modes; // output prediction rules pointers
int width, height; // frame dimensions

getImageSizeFromFile(pathToImg, &width, &height); // get frame dimensions
int sizeFrame = width * height; // calculate frame size
int sizeModes = (width/16) * (height/16) * sizeof(int); // calculate prediction rules array size

cudaMallocHost((void**)&h_frame, sizeFrame); // allocate host frame memory (if not having yet)
cudaMalloc((void**)&d_frame, sizeFrame); // allocate device frame memory
cudaMallocHost((void**)& h_ip16modes, sizeModes); // allocate host rules memory
cudaMalloc((void**)& d_ip16modes, sizeModes); // allocate device rules memory

loadImageFromFile(pathToImg, width, height, &h_frame); // load image to h_frame
cudaMemcpy(d_frame, h_frame, sizeFrame, cudaMemcpyHostToDevice); // copy host frame to device

dim3 blockSize = {16, 4, 1}; // configure CUDA block (16 threads per image block, 4 blocks in CUDA block)
dim3 gridSize = {width / 16, height / 64, 1}; // configure CUDA grid to cover the whole frame
intraPrediction16x16<<<gridSize, blockSize>>>(d_frame, d_modes_out); // launch kernel

cudaMemcpy(h_ip16modes, d_ip16modes, sizeModes, cudaMemcpyDeviceToHost); // copy rules back to host

cudaFree(d_frame); cudaFreeHost(h_frame); // free frames memory
cudaFree(d_ip16modes); cudaFreeHost(h_ip16modes); // free rules memory
```

SAD and SATD

- Sum of Absolute Transformed Differences increases MVF quality
- More robust than SAD
- Takes more time to compute
- Can be efficiently computed, i.e. split 16x16 block into 16 blocks of 4x4 pixels and do transform in-place

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Motion Estimation

Motion Estimation

Motion Estimation

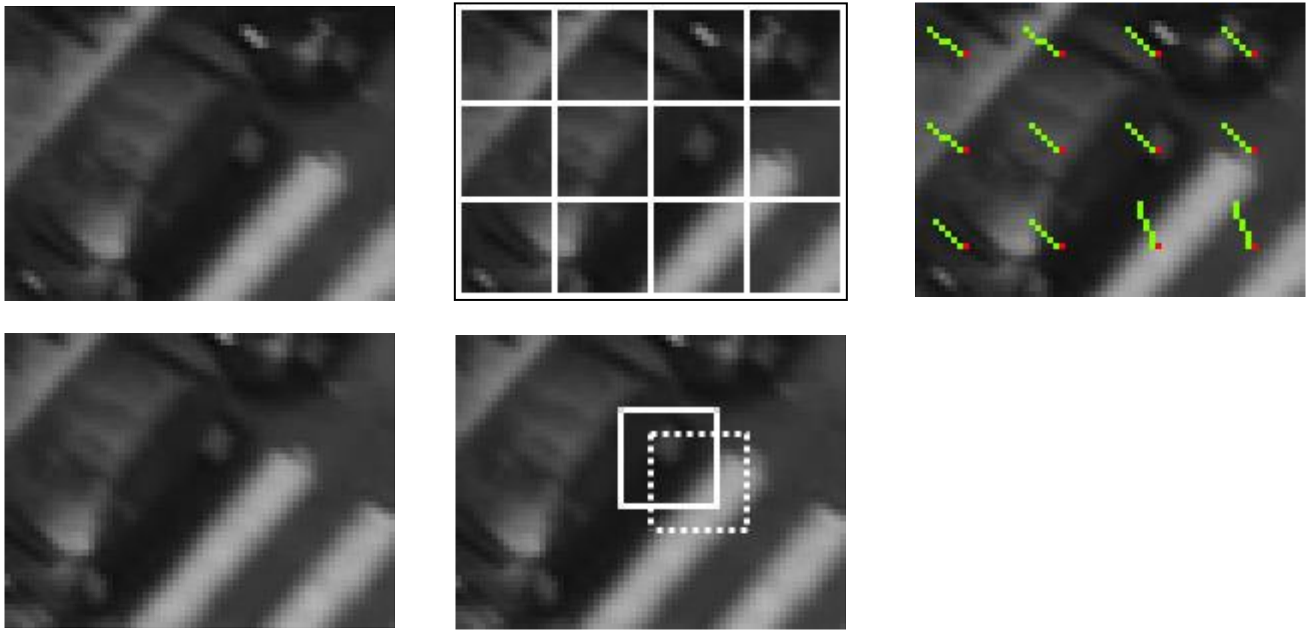
Motion Estimation

Like in Intra Prediction, when doing block-matching Motion Estimation a frame is split into blocks (typically 16x16, 8x8 and subdivisions like 16x8, 8x16, 8x4, 4x8, 4x4)

Original | Block match | Motion Vector Field

Current (P) frame

Reference (I, previous) frame

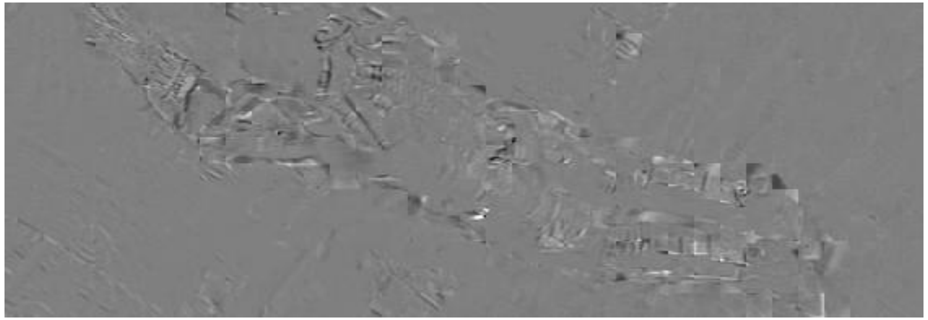


Motion Estimation

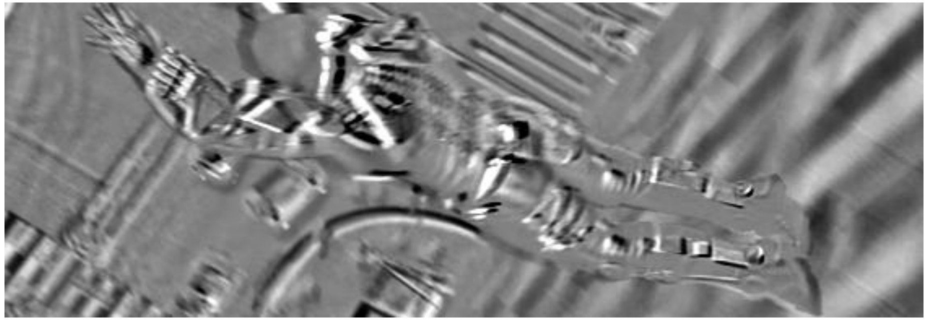
Motion vectors field (MVF) quality can be represented by the amount of information in the residual (the less – the better)

Original frame

Residual Original-Compensated



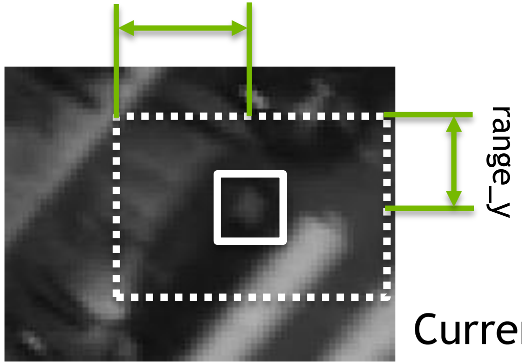
Compensated from reference (notice blocking)



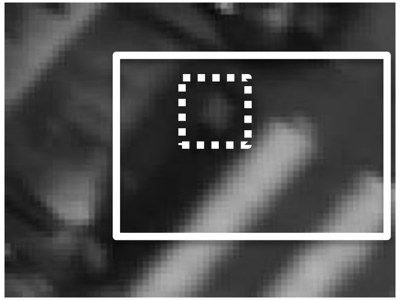
Residual Original-Reference

ME – Full Search

- Full search of the best match in some local area of the frame
- The area is defined as a rectangle centered on the block that we want to predict and going range_x pixels both sides along X axis and range_y pixels along Y axis
- range_x is usually greater than range_y because horizontal motion prevails in movies
- Very computationally expensive though straight-forward
- Doesn't "catch" long MVs outside of the range



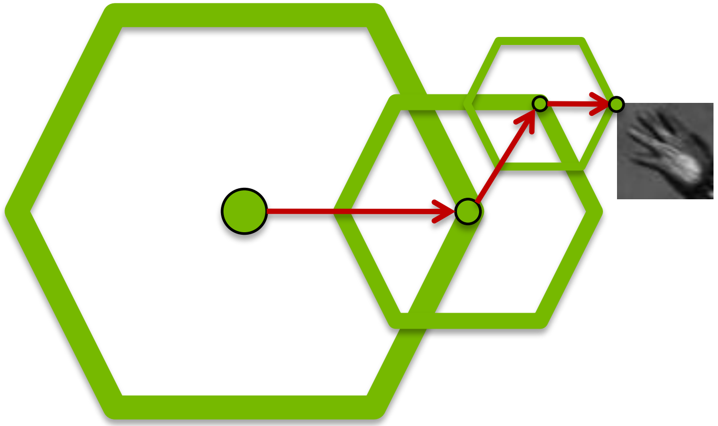
Current frame



Reference frame

ME – Diamond search

- Template methods class reduces the amount of block matches by the order of magnitude comparing to Full Search
- On every iteration the algorithm matches 6 blocks with anchors that form diamond centered on the initial position
- The diamond size doesn't grow and must reduce with the amount of steps made

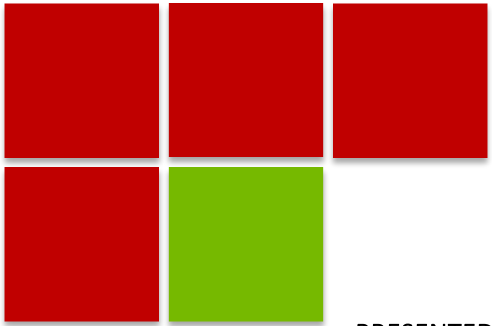


ME – Diamond search

- The majority of template methods are based on gradient descent and have certain application restrictions
- When searching for a match for the block  on the frame below, red initialization is not better than a random block selection, while green will likely converge
- Used often in junction with some other methods as a refinement step

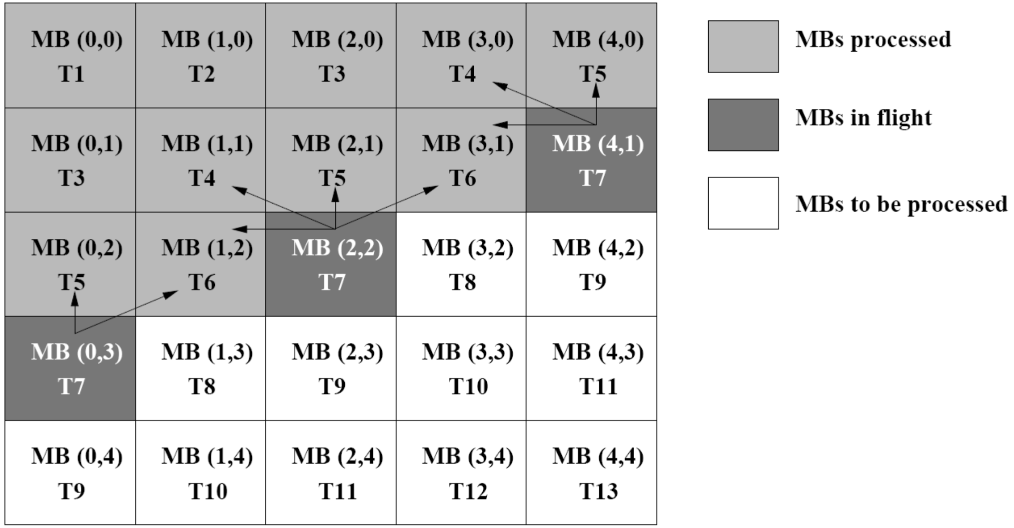
ME – Candidate Set

- Objects that move inside a frame are typically larger than a block, adjacent blocks have similar motion vectors
- Candidate set is a list of motion vectors that have to be checked in first place. The list is usually filled with motion vectors of already found adjacent (spatially or temporarily) blocks
- The majority of CPU-school ME algorithms fill candidate set with motion vectors found for left, top-left, top, top-right blocks (assuming block-linear YX-order processing)
- This creates an explicit data dependency which hinders efficient porting of the algorithm to CUDA



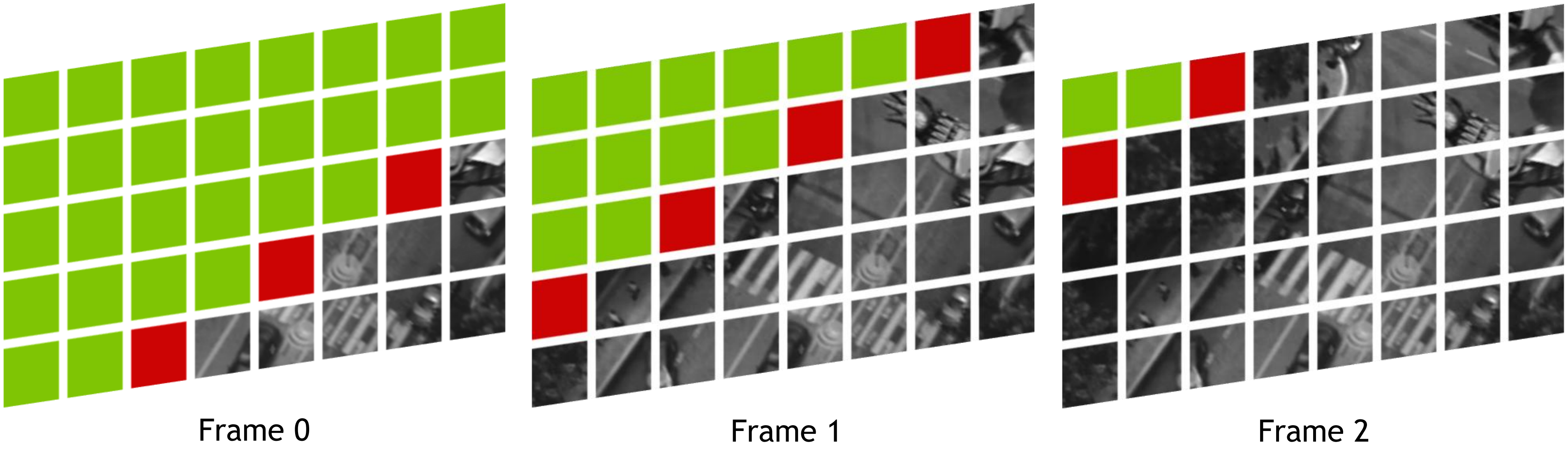
ME – 2D-wave

- A non-insulting way to resolve spatial data dependency
- Splits processing into the list of groups of blocks, with possibility of parallel processing of blocks within any group. Groups have to be executed in order
- The basic idea – to move by independent slices from the top-left corner to the down-right. TN stands for the group ID (and an iteration)
- Inefficiency of CUDA implementation:
  - Low SMs load
  - Need work pool with producer\consumer roles, or several kernel launches
  - Irregular data access



ME – 3D-wave

- 3D-wave treats time as a 3^{rd} dimension and moves from the top-left corner of the first frame (cube origin) by groups of independent blocks
- Useful for resolving spatio-temporal candidate set dependencies



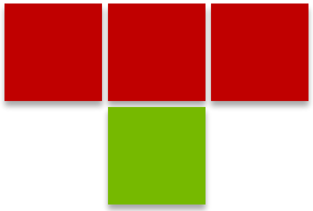
Legend
- Blocks with MV found
- Blocks to be processed (in parallel)

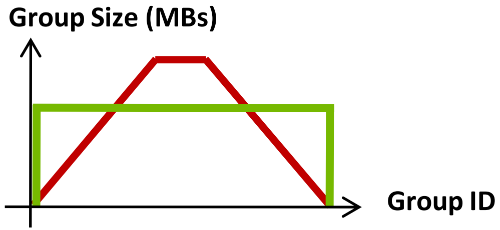
ME – GPU-friendly candidate set

- 2D and 3D wave approaches make GPU mock CPU pipeline by using clever tricks, which rarely agree with best GPU programming practices
- If full CPU-GPU data compliance is not a case, then revising the candidate set or even the full Motion Estimation pipeline is necessary
- Revised GPU-friendly candidate sets might give worse MVF. But they will free tons of time spent on maintaining useless structures, which in turn can be partially spent on doing more computations to leverage MVF quality

ME – GPU-friendly candidate set

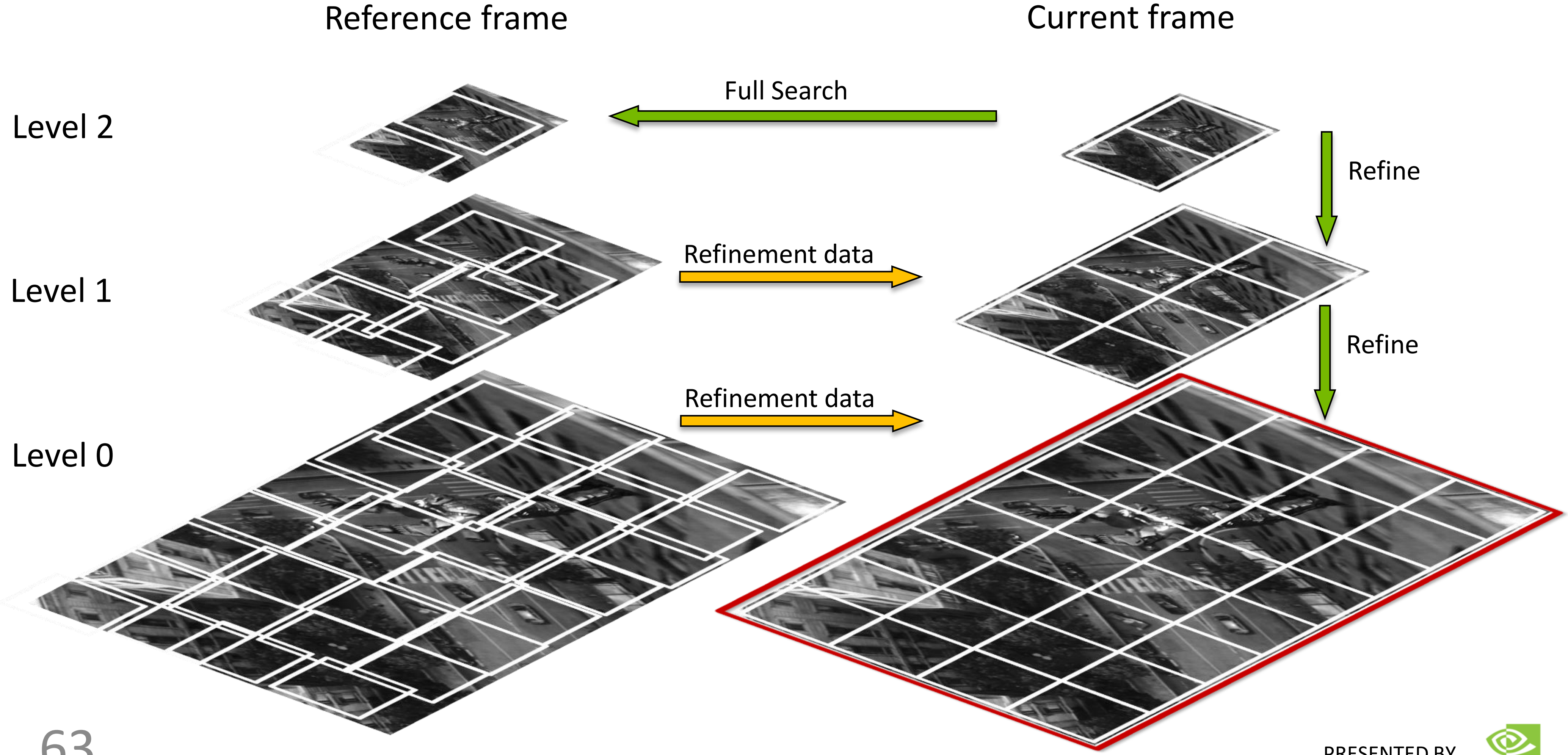
- **Any candidate set is friendly if it is known for every block at the kernel launch time (see Hierarchical Search)**
- A candidate set with spatial MV dependency can be formed as below
- Just by removing one (left) spatial candidate, we:
  - Enhance memory access pattern
  - Relax constraints on work producing\consuming
  - Reduce the code size and complexity
  - Equalize parallel execution group sizes





- Can be implemented in a loop where each CUDA block processes a column of the frame of few image blocks width
- Some grid synchronization trickery is still needed

ME – Hierarchical Search



ME – Hierarchical Search

Algorithm:

1. Build image pyramids
2. Perform full search on the top level
3. A motion vector of level K is the best approximation (candidate) for all “underlying” blocks on level K+1 and can be in candidate set for block on all subsequent levels
4. MVs refinement can be done by any template search
5. On the refinement step, N best matches can be passed to subsequent levels to prevent the fall into the local minima

GPU TECHNOLOGY CONFERENCE

ME – Hierarchical Search

Hierarchical search ingredients:

1. Downsampling kernel (participates in the creation of pyramid)
2. Full search kernel (creates the initialization MVF on the largest scale, can be used to refine problematic areas on subsequent layers)
3. Template search kernel (refines the detections obtained from the previous scales, or on the previous steps, frames)
4. **Global** memory candidate sets approach (to store N best results from the previous layers and frames)

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

CUDA and OpenCL

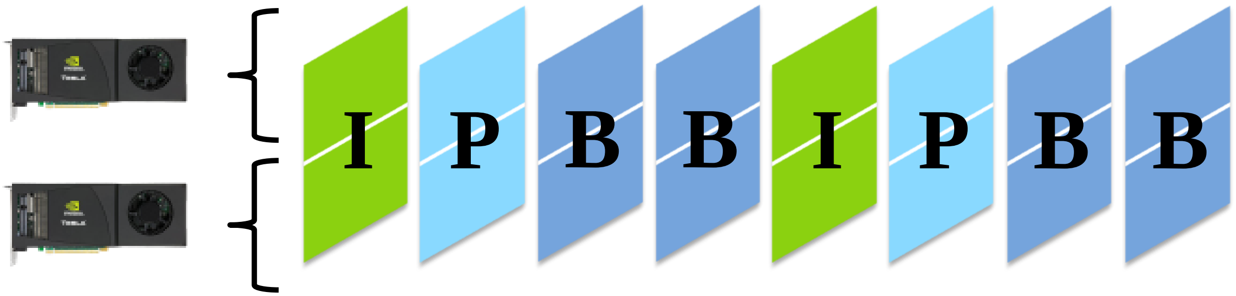
- For the sub-pixel Motion Estimation precision CUDA provides textures with bilinear interpolation on access
- CUDA provides direct access to GPU intrinsics
- CUDA: C++, IDE (Parallel Nsight), Visual Studio Integration, ease of use
- OpenCL kernels are not a magic bullet, they need to be fine-tuned for each particular architecture
- OpenCL for CPU vs GPU will have different optimizations

Tesla and Fermi architectures

- All algorithms will benefit from L1 and L2 caches (Fermi)
- Forget about SHMEM bank conflicts when working with 1-byte data types
- Increased SHMEM size (Tesla = 16K, Fermi = up to 48K)
- 2 Copy engines, Parallel kernel execution

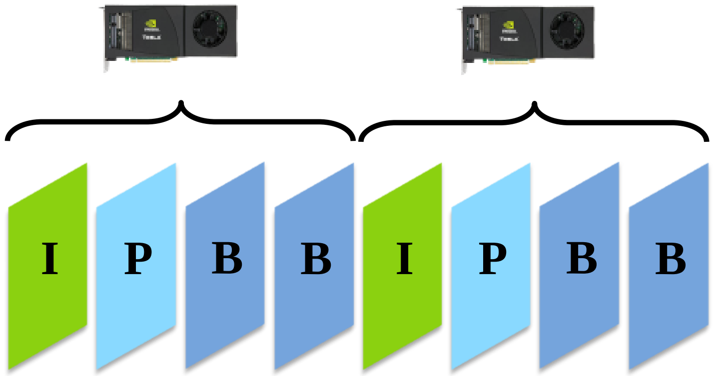
Multi-GPU encoding

- One slice per GPU
  - Fine-grain parallelism
  - Requires excessive PCI-e transfers to collect results either on CPU or on one of the GPUs to generate the final bitstream
  - Requires data frame-level data consolidation after every kernel launch
  - Data management might get not trivial due to apron processing



Multi-GPU encoding

- N GOPs for N GPUs
  - Coarse-grain parallelism
  - Better for offline compression
  - Higher utilization of GPU resources
  - Requires N times more memory on CPU to handle processing
  - Drawbacks: must be careful when concatenating GOP structures, CPU multiplexing may become a bottleneck,
  - Requires N times more memory on the host to process video



Other directions

- JPEG2000 - **cuj2k** is out there
  - open-source: http://sourceforge.net/projects/cuj2k/
  - needs more BPP for usage in the industry, currently supports only 8bpp per channel
- CUDA JPEG encoder – now available in CUDA C SDK
- MJPEG

Outline

- Motivation
- Video encoding facilities
- Video encoding with CUDA
  - Principles of a hybrid CPU/GPU encode pipeline
  - GPU encoding use cases
  - Algorithm primitives: reduce, scan, compact
  - Intra prediction in details
  - High level Motion Estimation approaches
  - CUDA, OpenCL, Tesla, Fermi
  - Source code

Source code

- The reference Full Search algorithm is implemented in CUDA and is available here in public domain:

  http://tinyurl.com/cuda-me-fs

  http://courses.graphicon.ru/files/courses/mdc/2010/assigns/assign3/MotionEstimationCUDA.zip

- Feel free to email us to see if we have an algorithm you need implemented in CUDA

Recommended GTC talks

We have a plenty of Computer Vision algorithms developed and discussed by our engineers:

– James Fung, "Accelerating Computer Vision on the Fermi Architecture"

– Joe Stam, "Extending OpenCV with GPU Acceleration"

– Timo Stich, "Fast High-Quality Panorama Stitching"

Ideas? Questions? Suggestions?

reach me via email:

Anton Obukhov <aobukhov@nvidia.com>