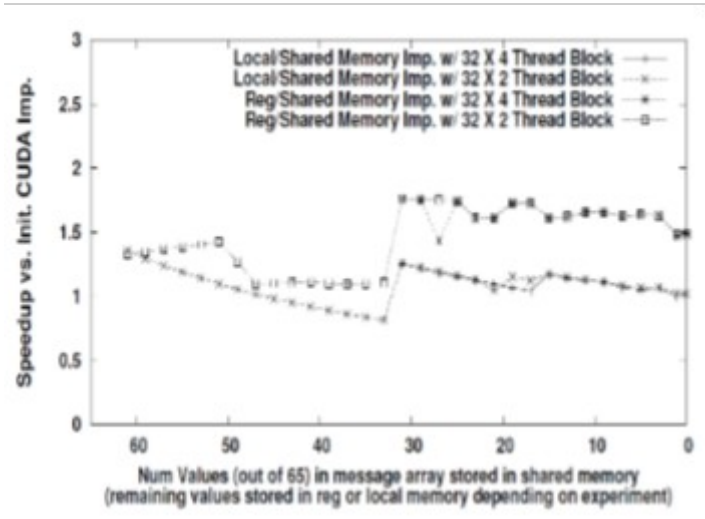
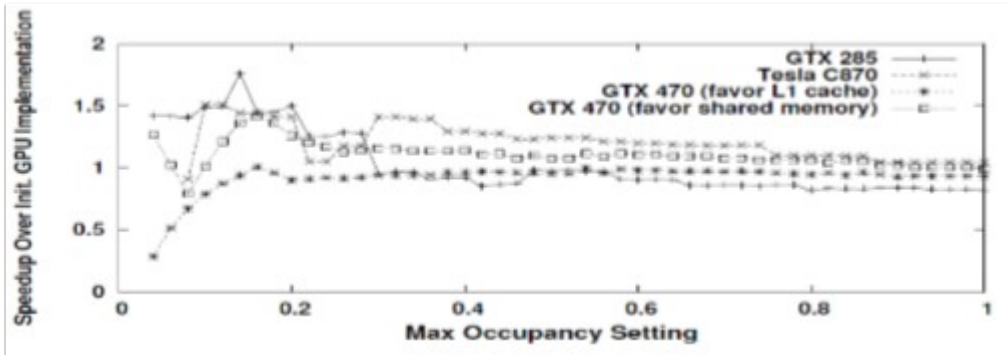
The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)

Optimizing and Auto-Tuning Belief Propagation on the GPU

Scott Grauer-Gray and Dr. John Cavazos

Computer and Information Sciences, University of Delaware





GPUs/CUDA

GPU: specialized hardware to parallel processor...

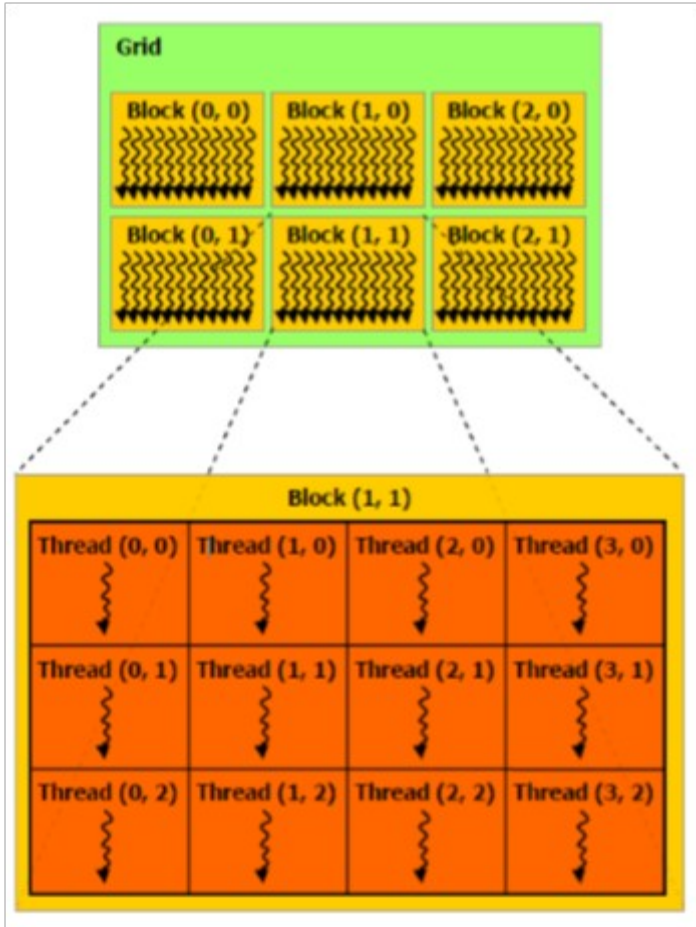
GPUs: more FLOPS and greater memory bandwidth than CPUs

nVidia encouraging usage of GPUs for general computing

CUDA Programming Model

Each thread has a unique ID in a grid of thread blocks

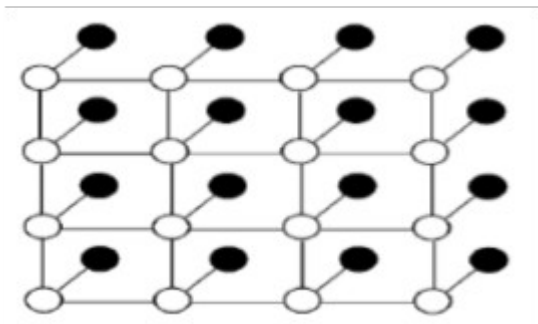
In theory, each thread processes the instructions of the kernel in parallel

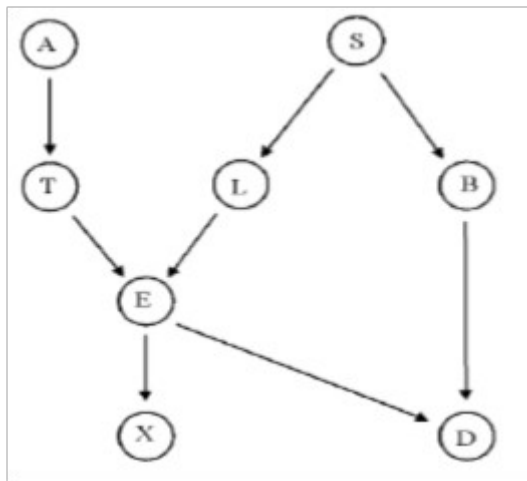


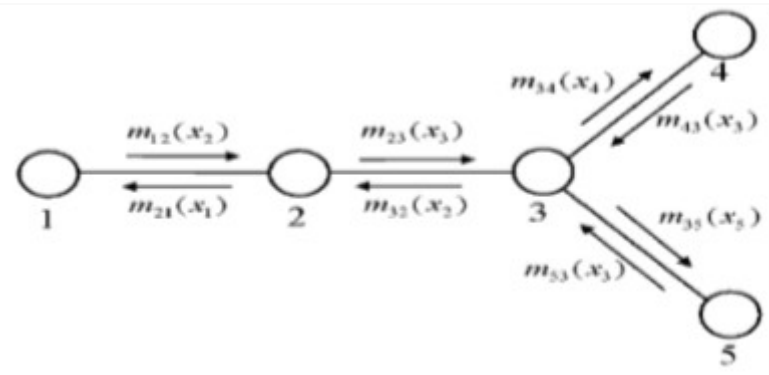
CUDA programming model

Belief Propagation

- General-purpose iterative algorithm for inference
- Applications in various disciplines
- Used for stereo vision problem in this work







Graphical representations: input to belief propagation implementations

Stereo Vision Problem

Input: scene images and disparity space

Goal: retrieve disparity between objects in both images

Output: Disparity at each pixel in input image

Below (from left to right): Tsukuba stereo set and resultant disparity map

Belief Propagation for Stereo Vision

Stereo algorithms using belief propagation give accurate results, but algorithm has shortcomings:

- Longer run-time vs. other stereo algorithms
- High storage requirements

Belief Propagation for stereo vision

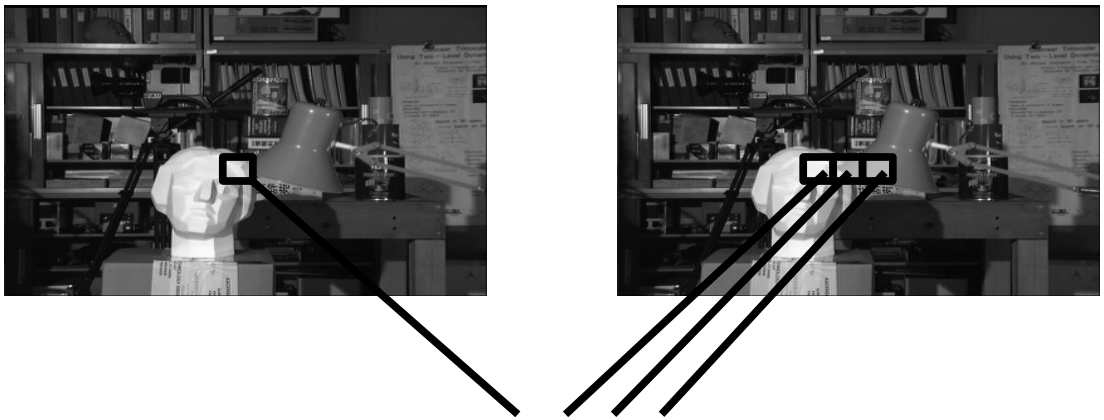
Previously implemented on the GPU:

- Using graphics API (example: OpenGL)
- Using CUDA architecture
- Implementations show decrease in running time versus CPU implementation
- **In this work, we optimize belief propagation on the GPU**

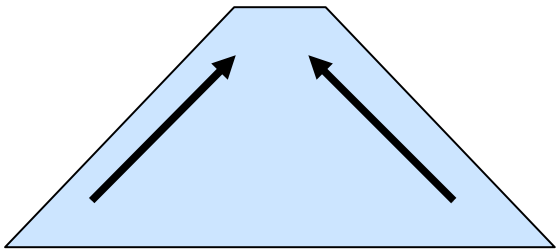
Belief Propagation for Stereo Vision

Consists of the following steps:

1. Compute data cost for each pixel at each disparity.



2. Iteratively compute data costs at each succeeding level.

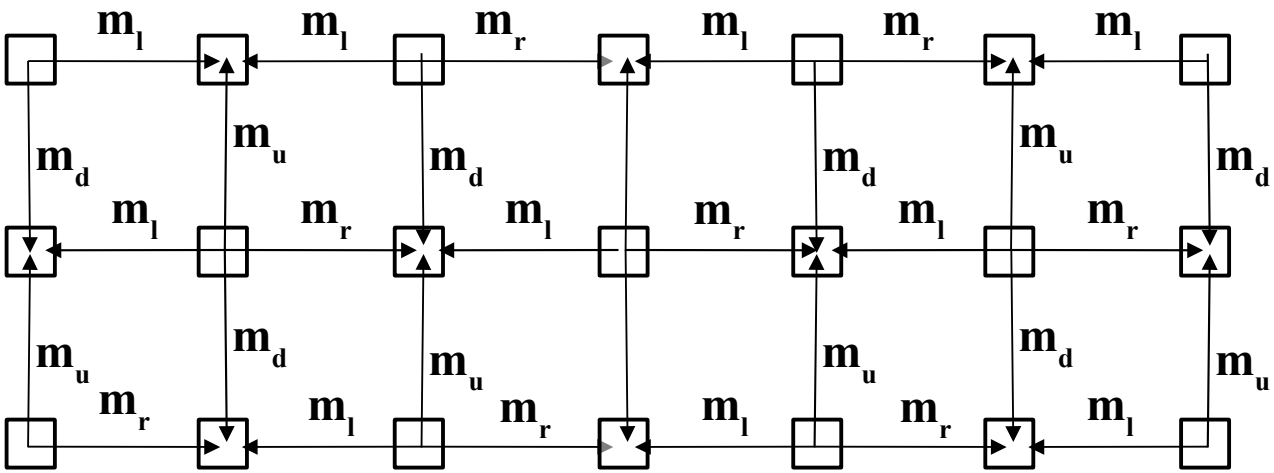


Belief Propagation for Stereo Vision

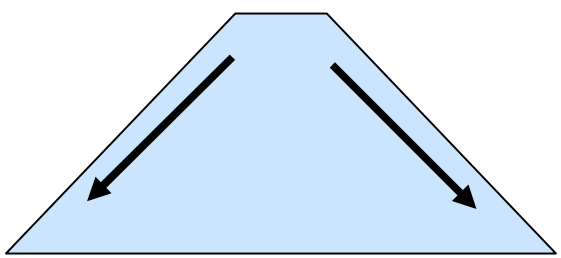
3. For each level in hierarchy:

a. Compute message values to send to neighboring pixels.

Repeat for i iterations, alternate between pixel sets



b. If not at bottom level, copy message values at each pixel to block of pixels in next level.



Belief Propagation for Stereo Vision

4. Retrieve disparity estimates at each pixel.

Optimizing belief propagation for stereo vision

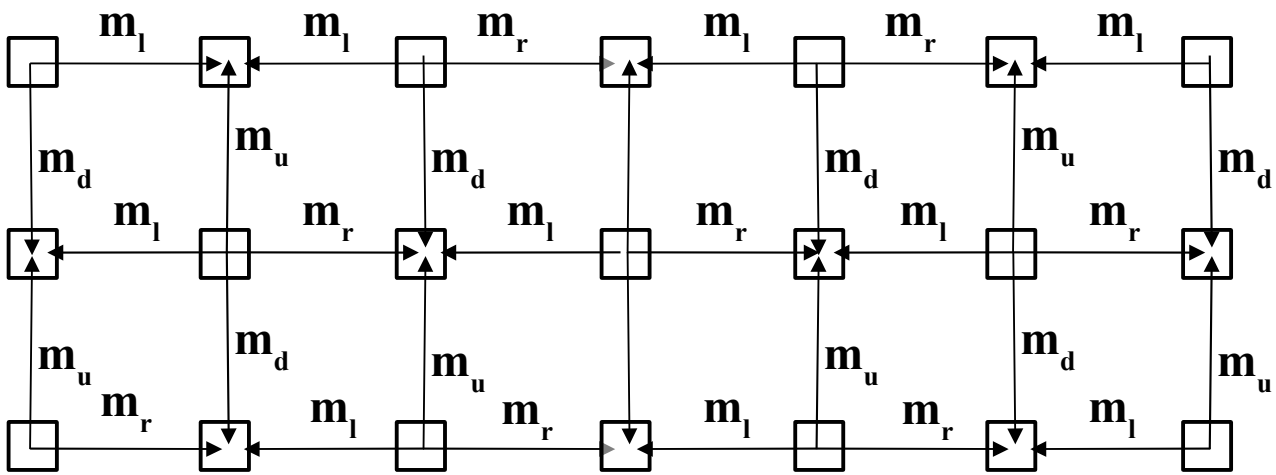
First step: Generate an initial implementation on the GPU:

Consists of the 5 steps/sub-steps

- Each parallelized on GPU using CUDA

70% of the running time is spent in step 3a

- Optimizations are performed on kernel for this step



Primary Aspects of Dominant Kernel

Many calculations performed on structure known as **message array**

- Array length equal to number of values in disparity space
- Array stored in local memory in initial implementation

Accesses to local memory on the GPU are slow, so...

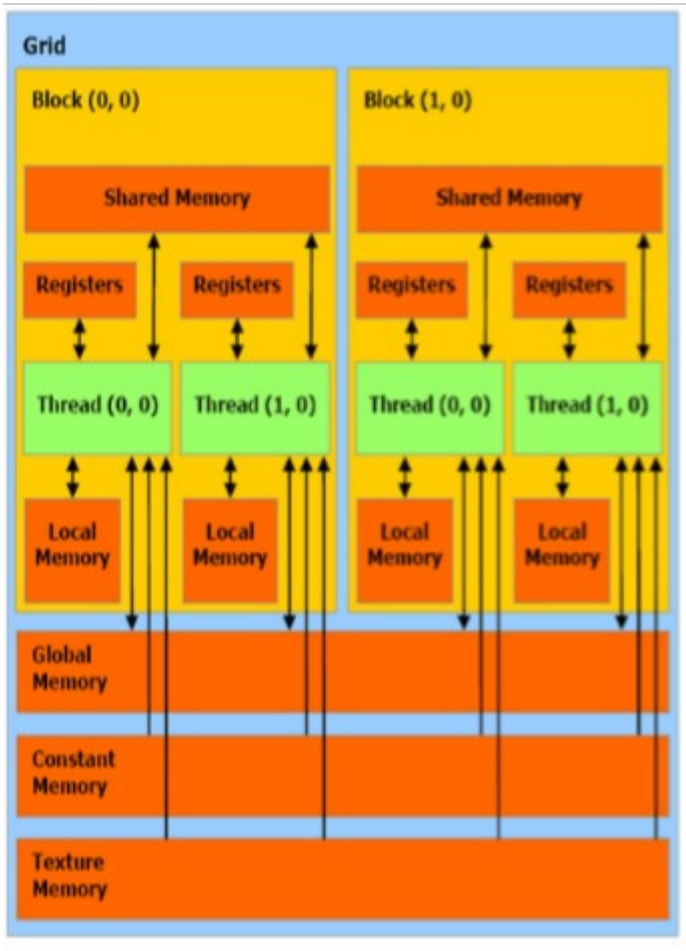
Might converting local memory accesses to register and/or shared memory accesses improve the running time?

Message array

Memory Hierarchy on the GPU

- Register memory and local memory is per thread
- Shared memory is per thread-block
- Limited supply of low-latency register and shared memory per multiprocessor

- **Utilize each form of memory in following implementations**



Code from initial implementation

```
float messageArray[N];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
        messageArray[i] = dataC[INDX_D_I] +
                neigh1[INDX_N1_i] +
                neigh2[INDX_N2_i] +
                neigh3[INDX_N3_i];

        if (messageArray[i] < currMin)
                currMin = m[i];
}
.
.
```

Code from shared memory implementation

```
__shared__ float messageArray_shared[N*THREAD_BLK_SIZE];

float currMin = INFINITY;

for (int i=0; i < N; i++)
{
        messageArray_shared[i_currThread] =
                dataC[INDX_D_I] +
                neigh1[INDX_N1_i] +
                neigh2[INDX_N2_i] +
                neigh3[INDX_N3_i];

        if (messageArray_shared[i_currThread] < currMin)
                currMin = m_shared[i_currThread] ;
}
.
.
```

Code from register memory implementation

```
float messageArray[N];

float currMin = INFINITY;

messageArray[0] = dataC[INDX_D_0] +
        neigh1[INDX_N1_0] +
        neigh2[INDX_N2_0] +
        neigh3[INDX_N3_0];

if (messageArray[0] < currMin)
        currMin = messageArray[0];
.
.
messageArray[N-1] = dataC[INDX_D_(N-1)] +
        neigh1[INDX_N1_(N-1)] +
        neigh2[INDX_N2_(N-1)] +
        neigh3[INDX_N3_(N-1)];

if (messageArray[N-1] < currMin)
        currMin = messageArray[N-1];
.
.
```

Results: initial and optimized implementations

Input: Tsukuba stereo set, 15 values in disparity space

Results on GTX 285

Storage / Loop Unroll Setting	Num Local Loads	Num Local Stores	Num Regs	Occup.	Total Running Time	Speedup from init. CUDA imp
Initial CUDA imp.	1307529	7360296	37	0.375	47.0 ms	- - -
Shared memory imp.	0	0	53	0.25	25.4 ms	1.85
Register memory imp.	0	0	110	0.125	24.3 ms	1.93

Initial vs. optimized results

Placing the **message array** in registers or shared memory improves results

- But...input Tsukuba stereo set has a relatively small disparity space

Research questions:
Might the programmer want to place split data between register and shared memory?

Might the programmer want the option to place data in local memory?

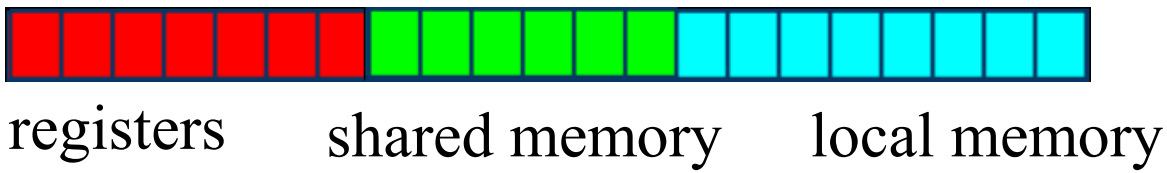
Our affirmative response to these questions represent the motivation for the following implementation...

Hybrid Implementation

Combines the register/shared/local memory options for data storage

Programmer can choose the amount of message array data placed in register, shared, and local memory

- **Greatly increases flexibility of optimization options...**



Hybrid Implementation

Input: `Cones' stereo set, disparity space of 65 values

Experiments use...

1. Shared/local memory w/ 32 X 4 thread block

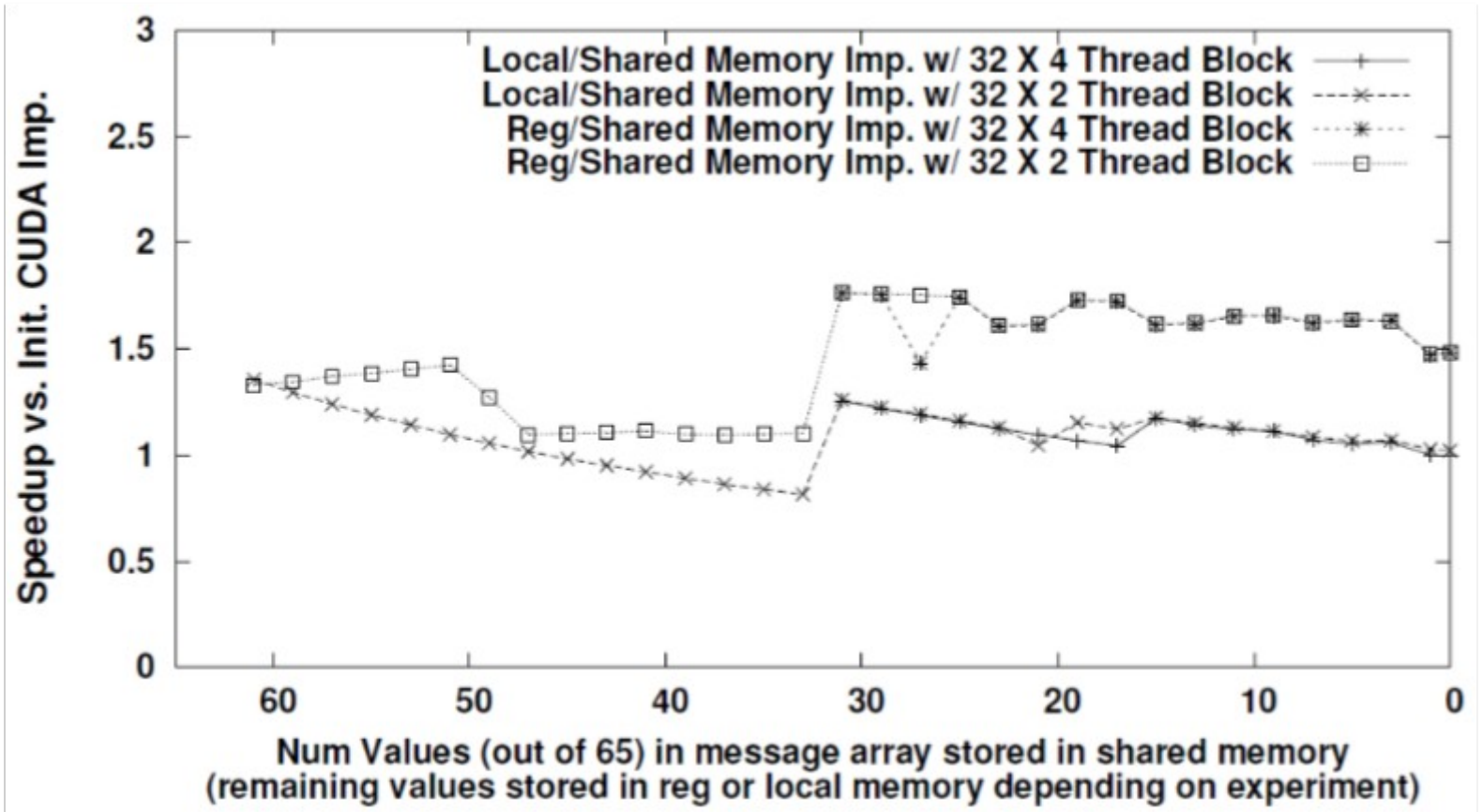
2. Shared/local memory w/ 32 X 2 thread block

3. Shared/register memory w/ 32 X 4 thread block

4. Shared/register memory w/ 32 X 2 thread block

Hybrid Implementation Results

Results on GTX 285:



Auto-tuning for optimization

Goal: develop auto-tuning framework to optimize algorithm across architectures / inputs

Hybrid config. results:

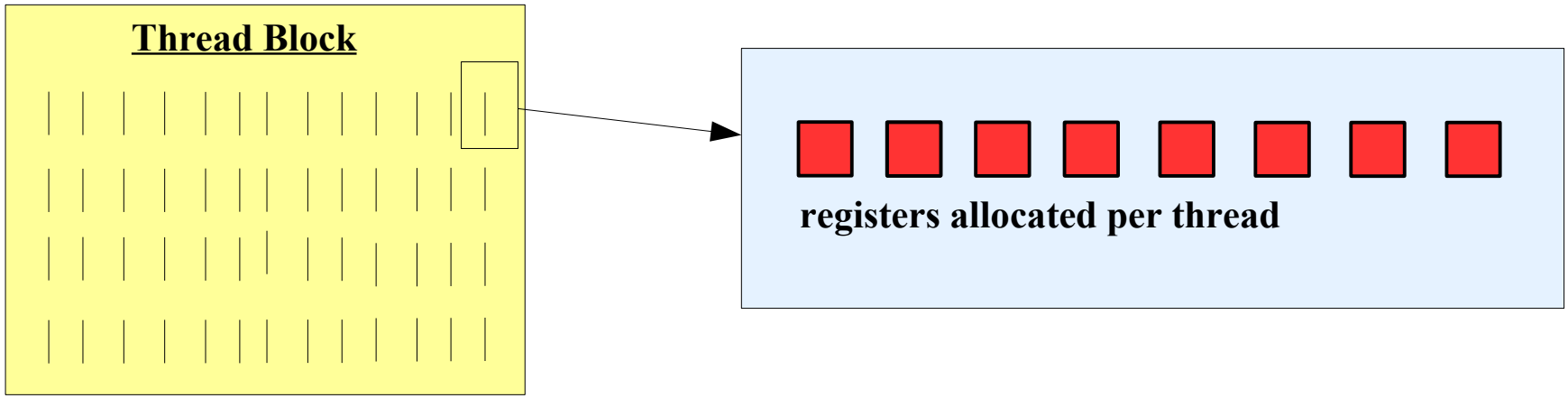
- Running time decreases when moving to a config. which allows for greater occupancy
- Running time then increases until the parameters allows for increase in occupancy...

We believe it is possible to retrieve a near-optimal configuration using a single carefully-generated configuration at selected occupancies

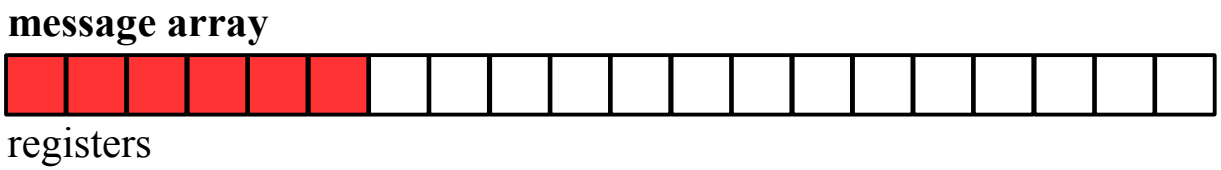
Auto-Tuning Framework

Input: stereo set, disparity space, max allowed occupancy

1. Determine the thread block dimensions and number of registers per thread

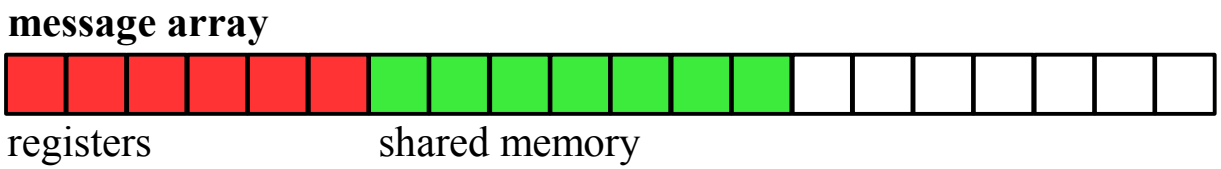


2. Place up to NUM_REG_VALUES_INITIAL values in register memory.

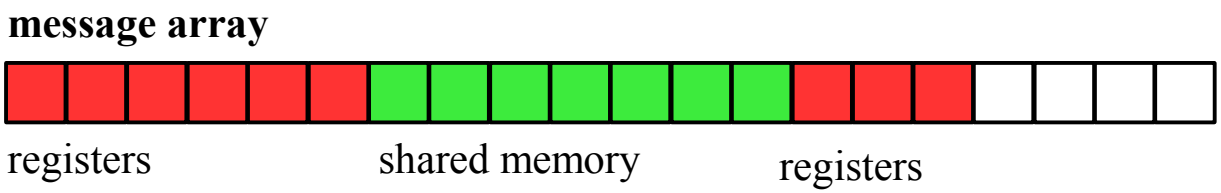


Auto-Tuning Framework

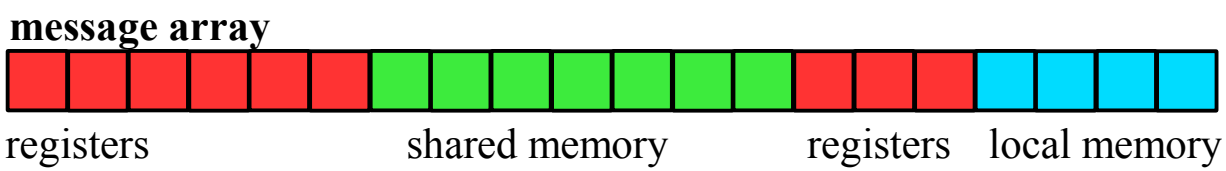
3. Determine and set the number of values to be stored in shared memory



4. Place remaining values in register memory; up to MAX_REG_VALUES total values in registers



5. Place any remaining values in local memory

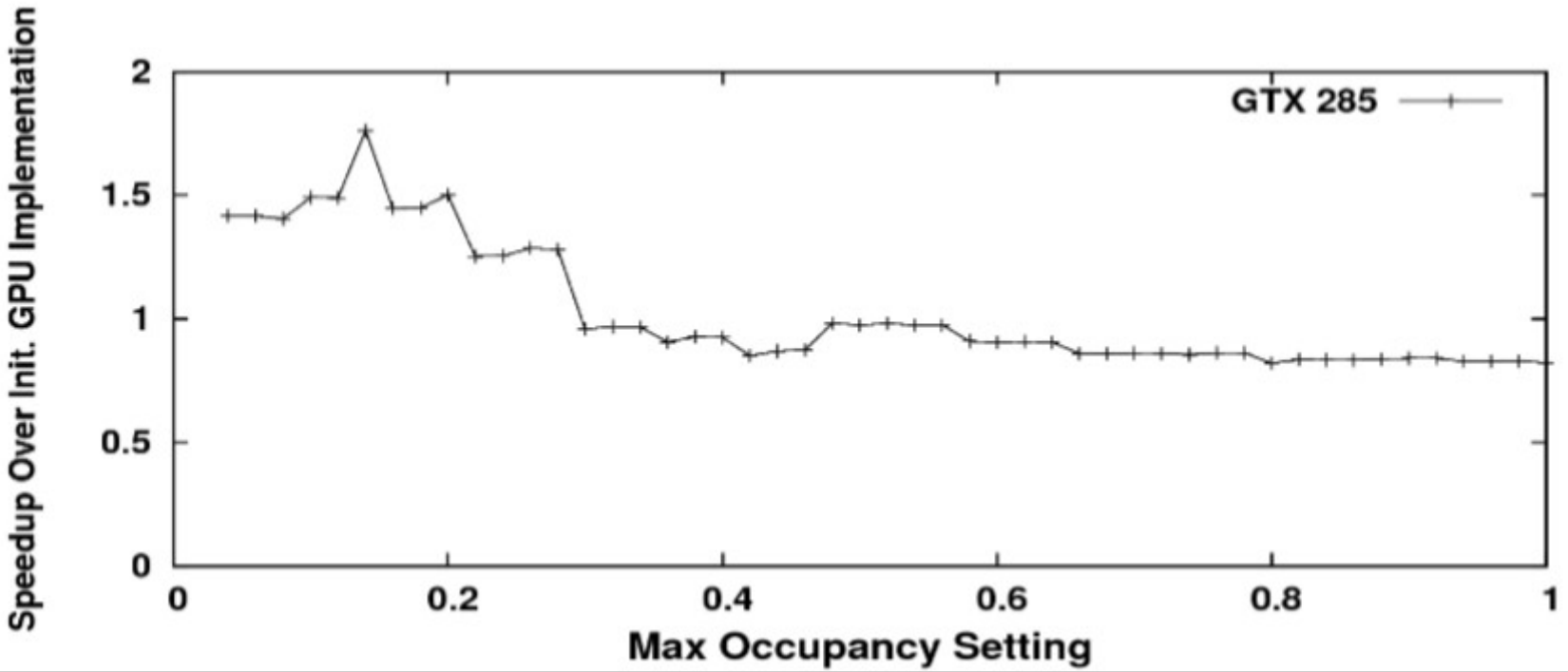


Auto-tuning experiments on GTX 285

Parameter values:

NUM_REG_VALUES_INITIAL = 12
MAX_REG_VALUES = 50

Results relative to initial implementation:



Experiments across GPUs

GTX 285
- 240 processors

Tesla C870
- 128 processors

GTX 470 (Fermi)
- 448 processors
- L1 cache

Results across GPUs

Hybrid implementation experiment results:

GPU (config.)	Thread block dimensions	Num vals in reg-shared-loc mem in mess array	Total Running Time	Speedup over init. CUDA imp.
GTX 285	32 X 2	34-31-0	238.0 ms	1.76
Tesla C870	32 X 2	14-51-0	840.2 ms	1.43
GTX 470 (favor L1 cache)	32 X 4	0-0-65	247.6 ms	1.00
GTX 470 (favor shared mem)	32 X 2	0-63-2	192.5 ms	1.39

Auto-tuning Result on each GPU

Optimal configuration obtained via auto-tuning

- Similar results to hybrid implementation experiments

GPU (config.)	**Max occupancy**	**Running Time (ms)**	**Speed-up over Init. CUDA imp.**	**% Difference in speedup from opt. config. in hybrid imp. experiments**
GTX 285	0.14	238.3 ms	1.76	0.0%
Tesla C870	0.10	797.7 ms	1.51	+5.06%
GTX 470 (favor L1 cache)	0.16	246.0 ms	1.01	+0.65%
GTX 470 (favor shared mem)	0.16	188.6 ms	1.42	+2.03%

Conclusions

- Improved running time of CUDA implementation
  - Took advantage of storage options in memory hierarchy
- Different GPU architectures require different optimizations
- Requires `general' framework where it is possible to optimize depending on the architecture