Think Silicon

Approximate Computing in Low Power GPUs

Georgios Keramidas

January 2015

Title: Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders

By: **Georgios Keramidas,** Chrysa Kokkala, and Iakovos Stamoulis

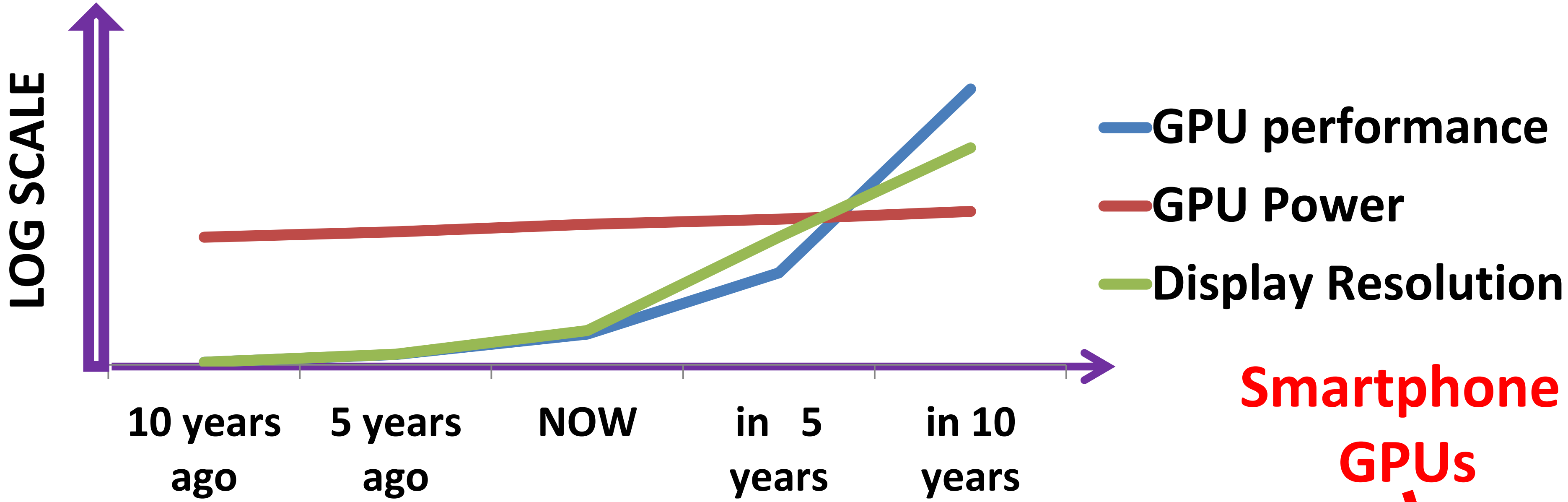
An EU-funded research project into low power GPU technology

SEVENTH FRAMEWORK PROGRAMME

Ultra Low Power GPUs for Wearables

- **Who are we?**
  - **Think Silicon is a privately held company founded in 2007**
- **What we do?**
  - **Design and Develop low power GPU IP semiconductor cores for mobile/embedded devices**
- **Market**
  - **Focus is the broader IoT and specifically the "Wearable" market**
- **Our mission**
  - **Support and collaborate with our customers to create mutual and enduring values in each phase of the project**

Moore's Law in Mobile GPUs



Smartphone GPUs

Applications → in next 5 years will need 4-5x the current GPU perf.

Display resolution → exponential increase (4K displays are here)

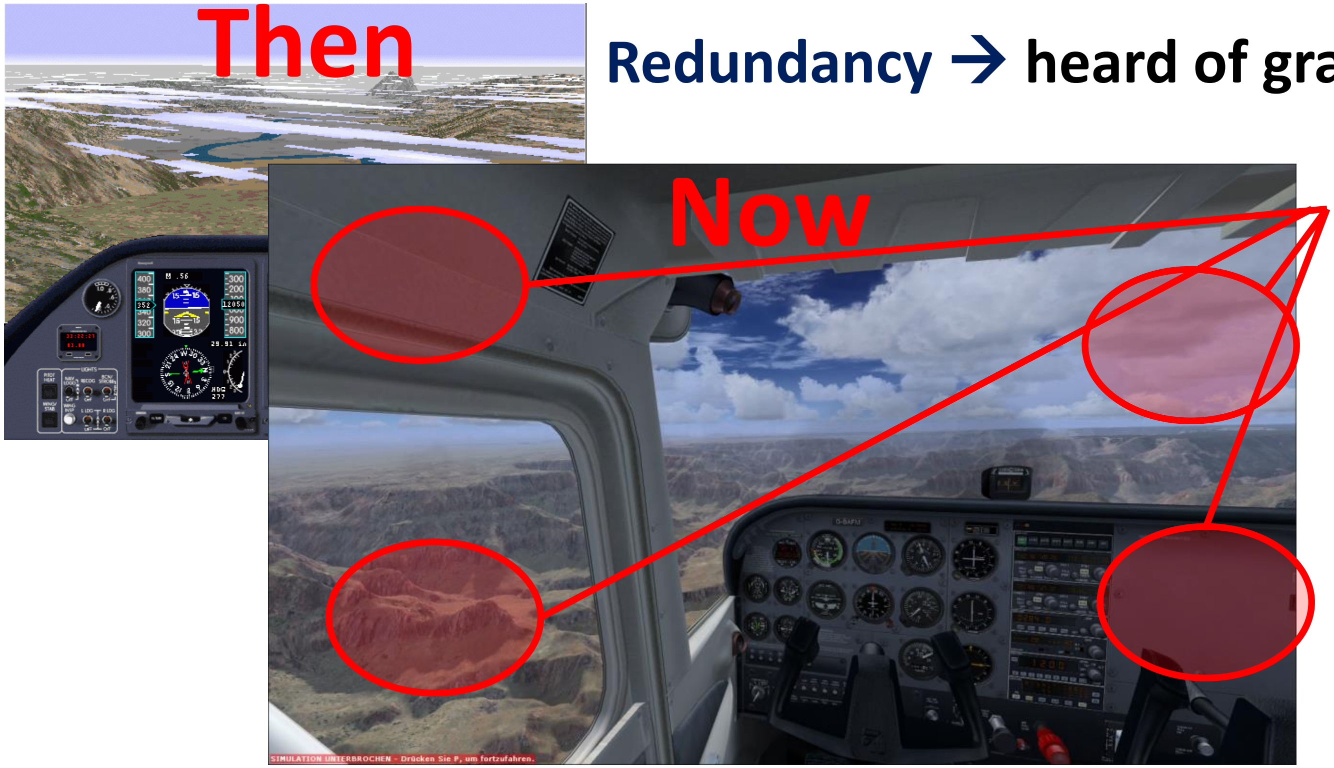
BUT… Power → roughly under the same power budget (few hundreds milliwatts)

IoT GPUs: few mWatts (< 3 mW) are devoted to graphics

New smart low power techniques must be found

Evolution in Graphics

Redundancy → heard of graphics applications



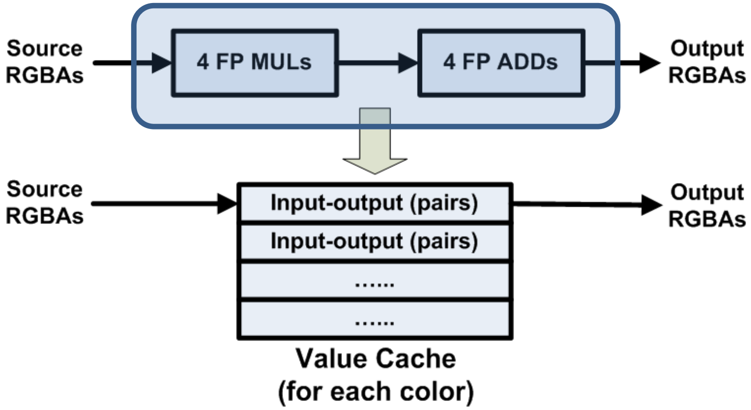
Many areas with the same or almost the same colours

Pixels with same colours → same calculations with same inputs ?

- If YES → we can reduce power by simply "remembering" the results from previous calculations

Remembering Previous Calculations ???

- Value Memoization or Value Reuse or Work Reuse or Value Cache
- Pros: Simple design → a memory array + some logic



- For each new pixel, first the Value Cache is checked using input colours
  - Match (value cache hit) → one local memory access, 8 less costly computations
  - Mismatch (value cache miss) → two local memory accesses (VC lookup + VC update) → extra power in the system

Memoization Failed in the Past !!!

- **Bad news: Value memoization is not able to pay off**
  - ~~Corbal et al. Fuzzy memoization for floating point multimedia applications~~
  - ~~Citron et al. Look It Up or Do the Math: An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization~~
  - ~~Huang et al. Exploiting Basic Block Value Locality with Block Reuse~~
  - ~~Richardson et al. Exploiting Trivial and Redundant Computation~~
  - ~~Bodik et al. Characterizing coarse-grained reuse of computation~~
  - ~~Sodani et al. Dynamic Instruction Reuse~~

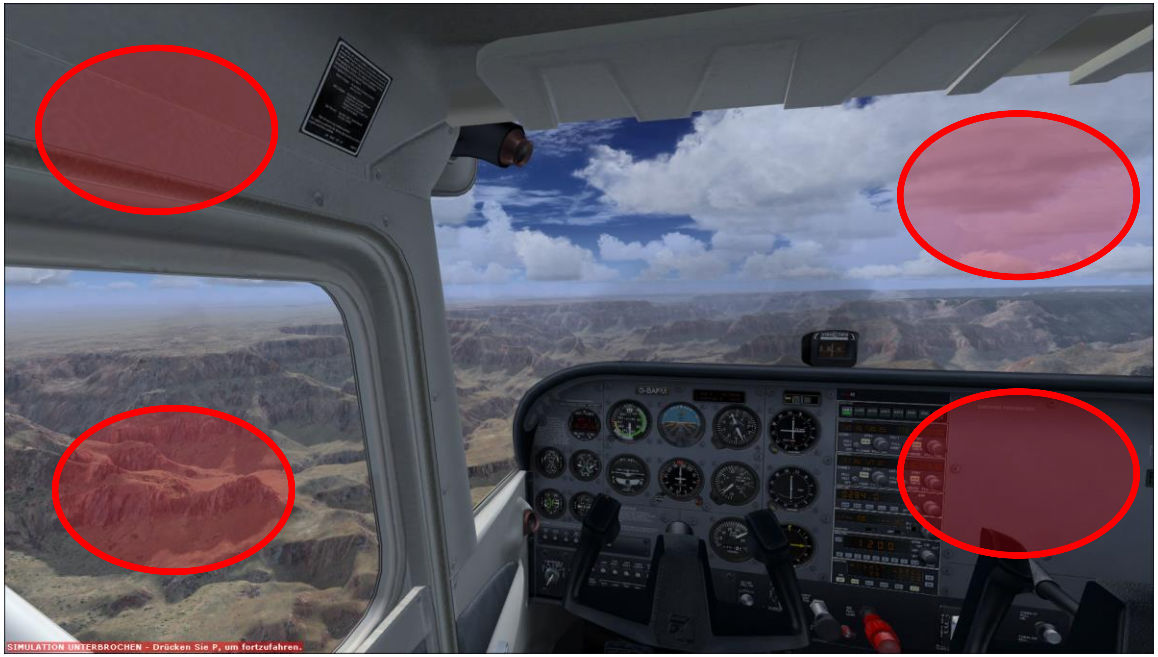
- **Reasons:**
  - **Very large reuse tables required**
  - **Redundancy or value reuses are limited**
  - **In CPU-like code, not enough number of blocks of (costly) instructions**

- **What about graphics?**
  - **Graphics apps and GPU architectures are promising?**

Memoization in Graphics ???

Graphics applications ??? YES

- Computer generated images → do have areas with similar colors
- Value caching will be beneficial

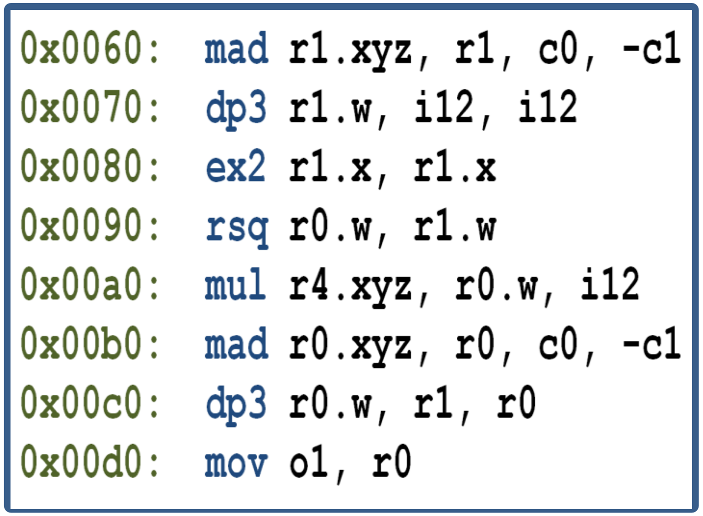


SIMULATION UNTERBROCHEN - Drücken Sie P, um fortzufahren.

Memoization in Graphics ???

GPU architecture ??? Definitely YES

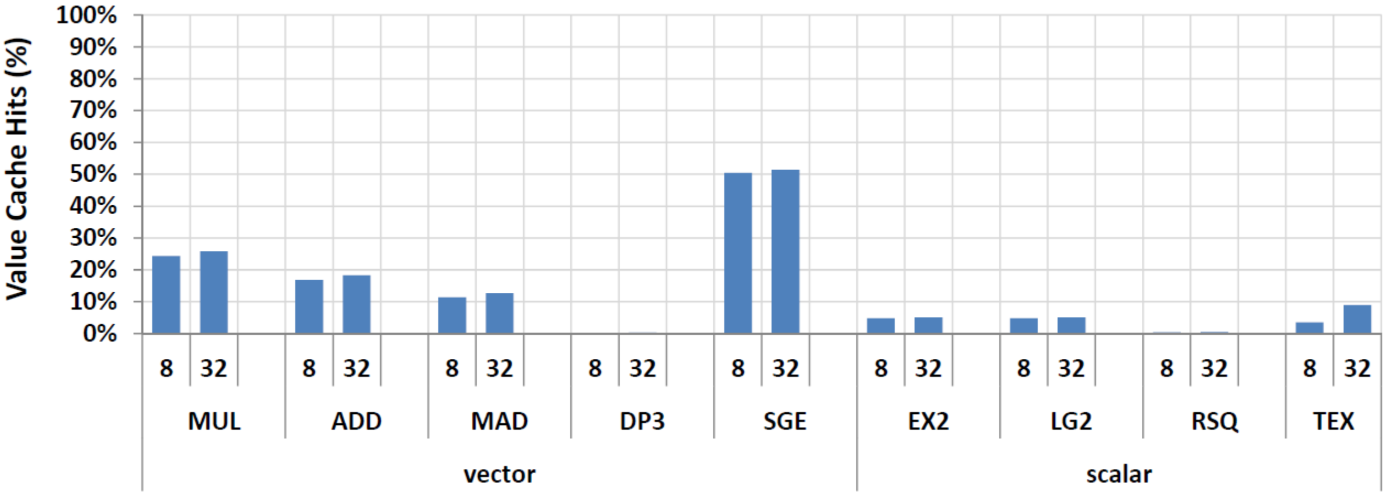
1. GPU code: data flow-like with a small number of registers per thread
2. GPU code: limited number of input registers, always one output register
3. GPU code: not conditional code (in most of the cases)
4. GPU code: typically 128 bits
5. GPU code: power hungry instructions e.g., log, rsq, or ex2
6. GPU code: many constant variables
   - No need to "remember" constants



```
0x0060:  mad r1.xyz, r1, c0, -c1
0x0070:  dp3 r1.w, i12, i12
0x0080:  ex2 r1.x, r1.x
0x0090:  rsq r0.w, r1.w
0x00a0:  mul r4.xyz, r0.w, i12
0x00b0:  mad r0.xyz, r0, c0, -c1
0x00c0:  dp3 r0.w, r1, r0
0x00d0:  mov o1, r0
```

A typical GLSL fragment shader

First Results

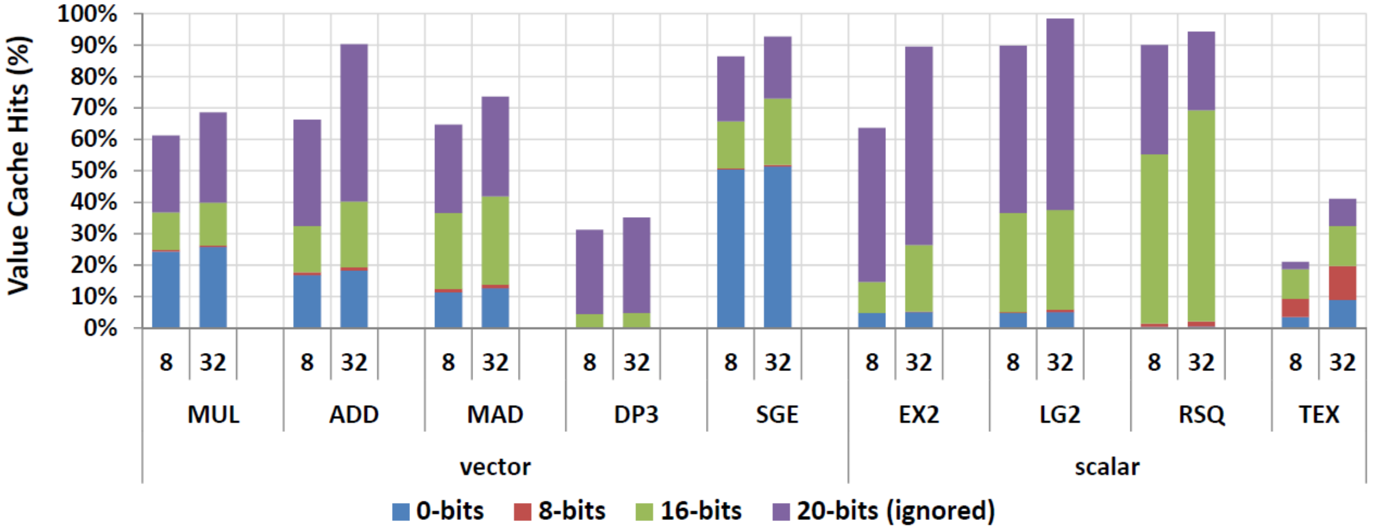


Average (0-bits):

- 12,96% 8-entries
- 14,21%, 32 entries

- Evaluation of Redundancy & Value Cache in all GPU instructions
  - Result 1: Value cache performance differs among the instructions depending on instruction type (scalar or vector and input registers)
  - Result 2: Redundancy is limited → only 14% on average in a 32 entries Value Cache

First Results using Approximate Matches



Average (0-bits):

- 12,96% 8-entries
- 14,21%, 32 entries

Average (8-bits):

- 16,04%, 32 entries

Average (16-bits):

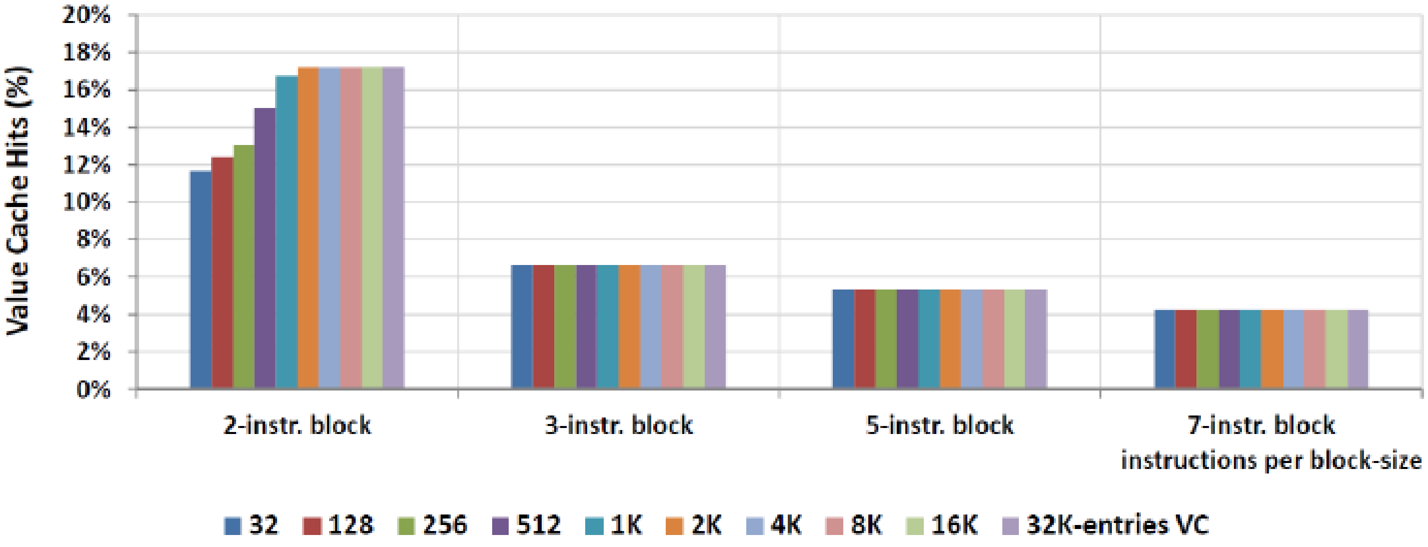
- 40,62%, 32 entries

Average (20-bits):

- 75,95%, 32 entries

- **Reduced Accuracy: from full VC matches to partial matches (partial matches: dynamically reduce the bits of mantissa)**
  - **Result: VC hit ratio increases exponentially → "accuracy" is the only viable way**

Value Cache in Instruction Groups

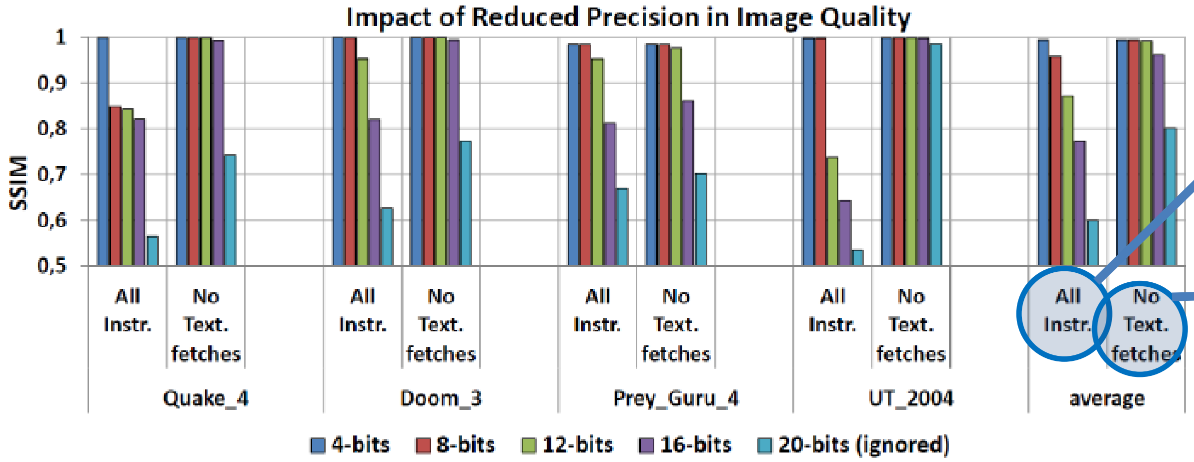


- **Evaluation of Value cache with full matches (Redundancy) in large memoization tables**
  - **Result: Redundancy cannot be captured with (unrealistically) large tables in groups of instructions**

What about Image Quality ?

- **Not all instructions must be equally precise**
  - **Result 1:** **Texture fetches in high precision**
  - **Result 2:** **Calculations in low precision**

More details can be found in the paper



Precision is reduced in all instructions

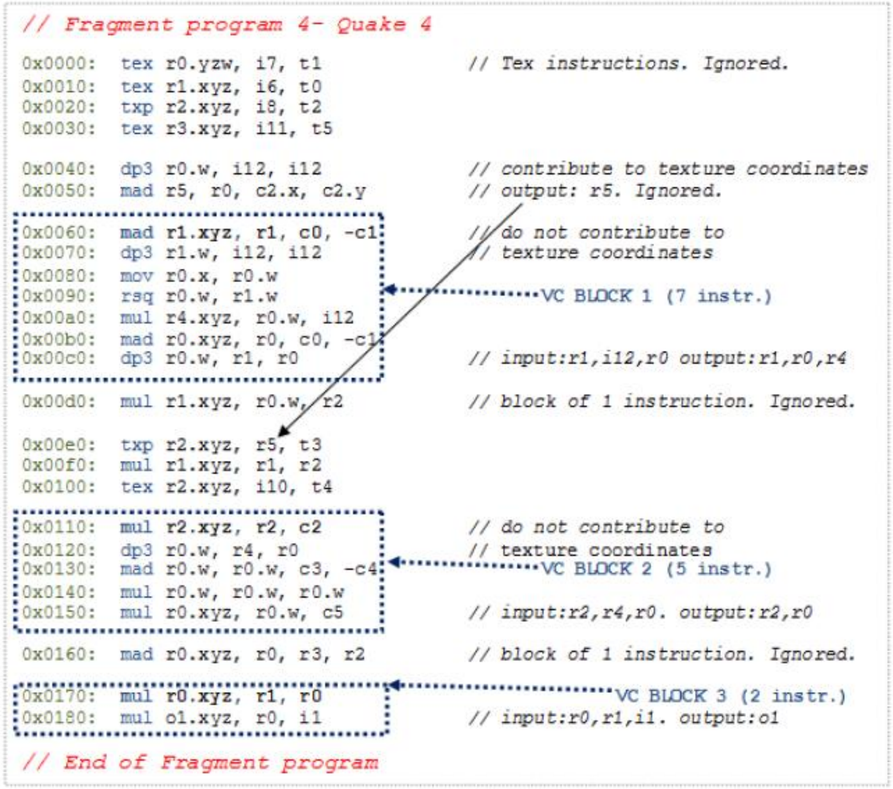
Precision is reduced only in arithmetic instructions (texture fetches in FULL precision)

- **Precision reduction in all instructions → NO more than 4-bits can be ignored during partial matches**
- **Precision reduction ONLY in arithmetic instructions → up to 16 bits can be ignored during partial matches**

Our Methodology

- VC instructions
  - AddEntries places new results in the VC in misses
  - LookupEntries retrieves from VC, in hits, or produces misses

- Putting all together → Compiler (LLVM)-level Methodology to automatically identify VC blocks in OpenGL fragment shaders



```
// Fragment program 4- Quake 4

0x0000:  tex r0.yzw, i7, t1              // Tex instructions. Ignored.
0x0010:  tex r1.xyz, i6, t0
0x0020:  txp r2.xyz, i8, t2
0x0030:  tex r3.xyz, i11, t5

0x0040:  dp3 r0.w, i12, i12              // contribute to texture coordinates
0x0050:  mad r5, r0, c2.x, c2.y          // output: r5. Ignored.

0x0060:  mad r1.xyz, r1, c0, -c1         // do not contribute to
0x0070:  dp3 r1.w, i12, i12              // texture coordinates
0x0080:  mov r0.x, r0.w
0x0090:  rsq r0.w, r1.w                  VC BLOCK 1 (7 instr.)
0x00a0:  mul r4.xyz, r0.w, i12
0x00b0:  mad r0.xyz, r0, c0, -c1
0x00c0:  dp3 r0.w, r1, r0                // input:r1,i12,r0 output:r1,r0,r4

0x00d0:  mul r1.xyz, r0.w, r2            // block of 1 instruction. Ignored.

0x00e0:  txp r2.xyz, r5, t3
0x00f0:  mul r1.xyz, r1, r2
0x0100:  tex r2.xyz, i10, t4

0x0110:  mul r2.xyz, r2, c2              // do not contribute to
0x0120:  dp3 r0.w, r4, r0                // texture coordinates
0x0130:  mad r0.w, r0.w, c3, -c4         VC BLOCK 2 (5 instr.)
0x0140:  mul r0.w, r0.w, r0.w
0x0150:  mul r0.xyz, r0.w, c5            // input:r2,r4,r0. output:r2,r0

0x0160:  mad r0.xyz, r0, r3, r2          // block of 1 instruction. Ignored.

0x0170:  mul r0.xyz, r1, r0              VC BLOCK 3 (2 instr.)
0x0180:  mul o1.xyz, r0, i1              // input:r0,r1,i1. output:o1

// End of Fragment program
```

Methodology (in a nutshell):

- Try to find the largest code segments (VC blocks) excluding texture fetch instructions or instructions than contribute to texture fetches
- Reasoning: in this way, the precision of the VC block can be aggressively reduced (increasing VC hits)

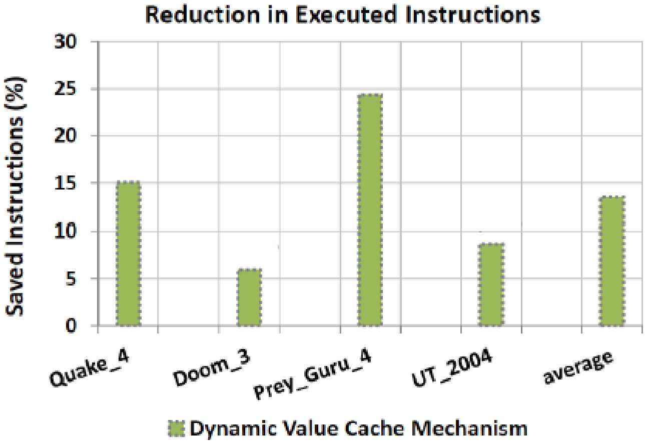
Results

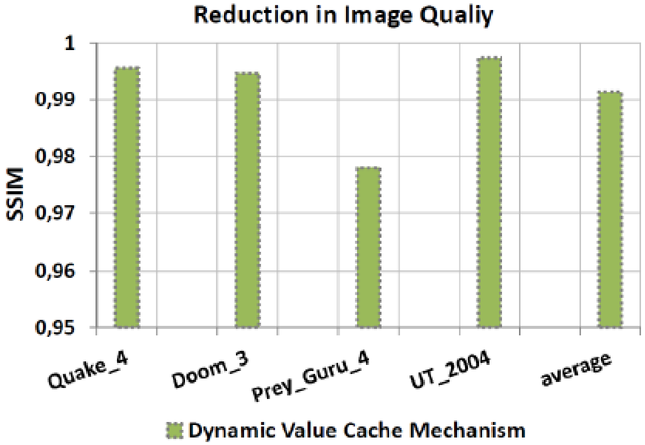
Frame	50	200	300
Quake_4	60	60	60
Doom_3	54	55	54
Prey_Guru_4	54	55	55
UT_2004	64	72	73
Average code coverage: 58.7%			

VC block selection methodology → 58.7% of fragment shaders code encapsulated in VC blocks

- **Dynamic Value Cache → Run-time, feedback-directed mechanism to control the interplay between precision reduction and QoS maximizing the value reuse benefits at the same time**

More details can be found in the paper

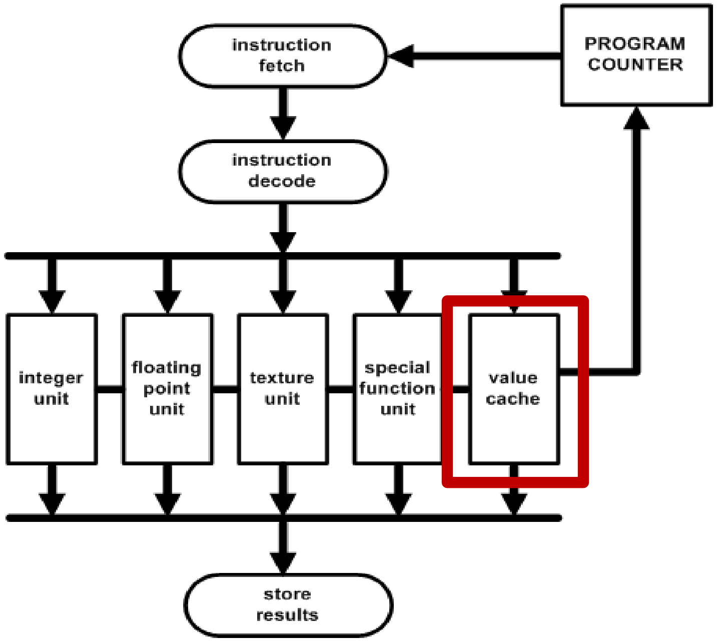


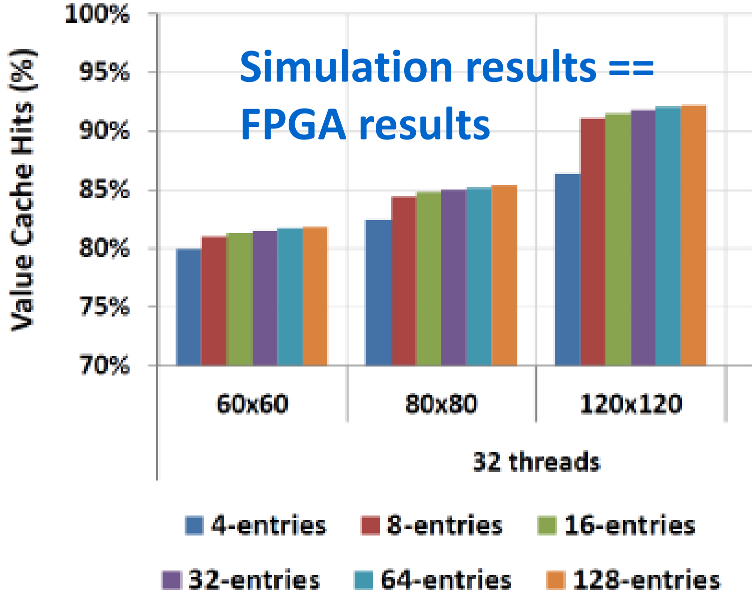


- **13.5% reduction in executed instructions**
- **>60% hits in value cache**
- **0.8% image quality loss**

Practical Issues

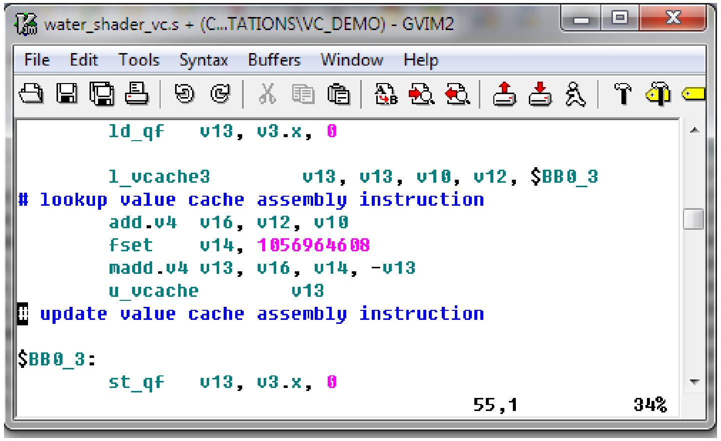
- Value Cache in silicon (in 2 company's products)
  - VC as an extra specialized functional unit in GPU data path
  - VCFU managed by machine instructions visible to GPU compiler/assembler
  - Extension of GPU ISA
  - VC instructions as LLVM intrinsic instructions
  - Insertion methodology implemented in the LLVM IR





Simulation results == FPGA results

Assembly Produced By LLVM



```
        ld_qf   v13, v3.x, 0

        l_vcache3       v13, v13, v10, v12, $BB0_3
# lookup value cache assembly instruction
        add.v4  v16, v12, v10
        fset    v14, 1056964608
        madd.v4 v13, v16, v14, -v13
        u_vcache        v13
# update value cache assembly instruction

$BB0_3:
        st_qf   v13, v3.x, 0
                                          55,1          34%
```

Conclusions

- Contribution: Value Cache mechanism
- Target: Remove redundant, complex arithmetic operations in OpenGL graphics applications
- VC strongly relies on the concept of approximate computing by reducing the accuracy of the value memoization comparisons in a dynamic fashion
  - Without using approximate computing techniques meager or negative benefits observed
- Overall: 13.5% reduction in executing instructions in modern fragment shaders with a negligible loss in the quality of the rendered images