Intel® C++ Compiler unveils compute power of Intel® Graphics Technology for general purpose computing

Anoop Madhusoodhanan Prabha

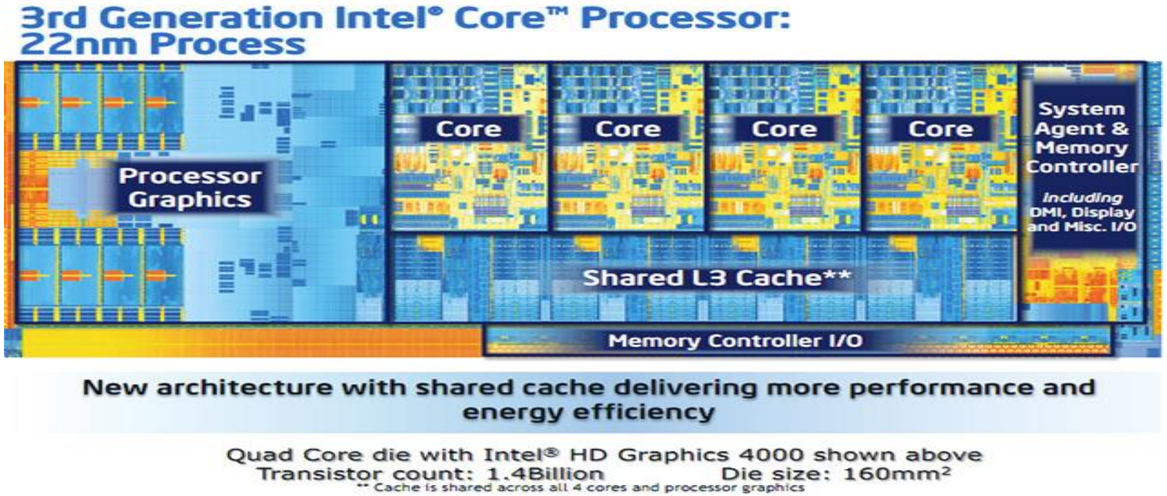
Agenda

- Why enable (Processor Graphics) GFX for general purpose computing?
- Preliminary Performance gain using GFX + CPU compute power
- Offload support
- GFX architecture
- Memory model
- Tuning applications for GFX
- Software Stack for Intel® C/C++ Compiler for GFX
- Intel® C++ compiler for GFX workflow
- Hardware and OS platforms supported
- Limitations
- Call to Action

Optimization Notice

Why enable GFX for general purpose computing?

- 3rd generation Intel® Core™ Processors and above have significant compute power (CPU + GFX)
- GFX occupies significant % of processor silicon area.
- GFX offload compiler helps full utilization of silicon area on the processor.
- Seamless porting experience by using Intel® Cilk™ Plus programming model.



Optimization Notice

Ease of Porting to GFX

Original Host code:

```
void vector_add(float *a, float *b, float *c){
        for(int i = 0; i < N; i++)
                c[i] = a[i] + b[i];
        return;
        }
```

Parallel Host code using Intel® Cilk™ Plus:

```
void vector_add(float *a, float *b, float *c){
        cilk_for(int i = 0; i < N; i++)
                c[i] = a[i] + b[i];
        return;
        }
```

Offloading function body to GFX:

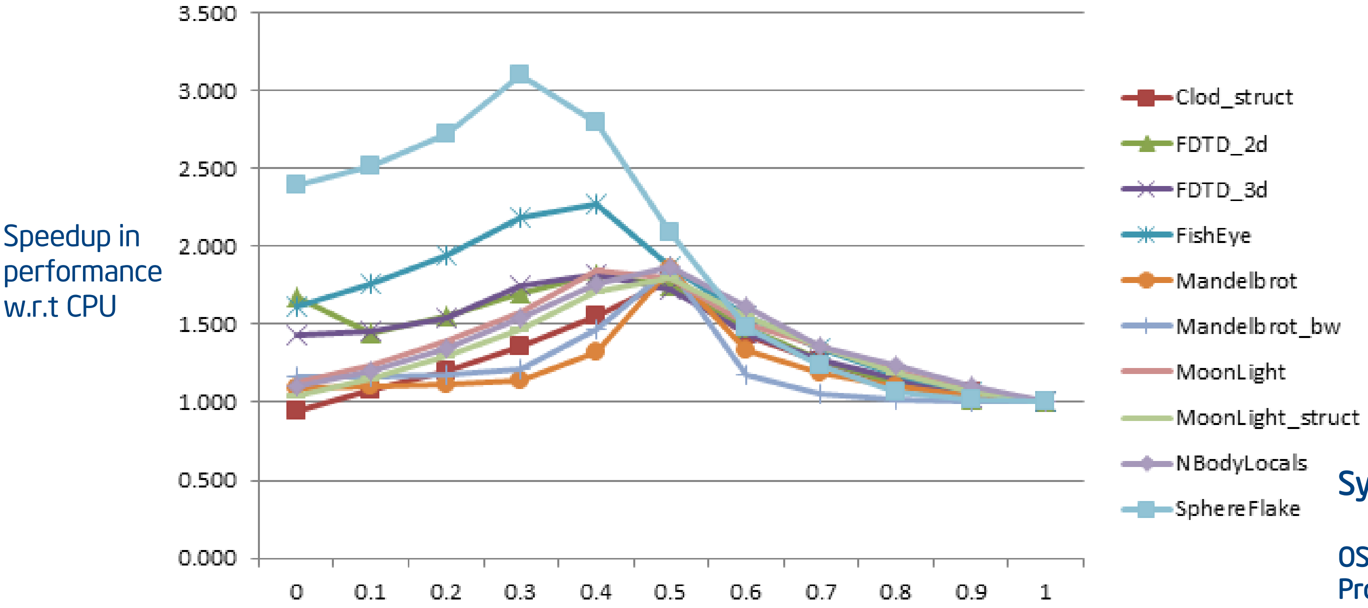
```
void vector_add(float *a, float *b, float *c){
        #pragma offload target(gfx) pin(a, b, c:length(N))
        cilk_for(int i = 0; i < N; i++)
                c[i] = a[i] + b[i];
        return;
        }
```

Creating GFX kernel for asynchronous offload:

```
__declspec(target(gfx_kernel))
        void vector_add(float *a, float *b, float *c){
        cilk_for(int i = 0; i < N; i++)
                c[i] = a[i] + b[i];
        return;
        }
```

Optimization Notice

Preliminary Performance gain using GFX + CPU compute power



Left to right -> Pure GPU execution to Pure CPU execution
0 – Pure GPU execution mode , 1- Pure CPU execution mode
0.1 – 10% workload on CPU and 90% on GPU
0.9 – 10% workload on GPU and 90% on CPU

System Specification:

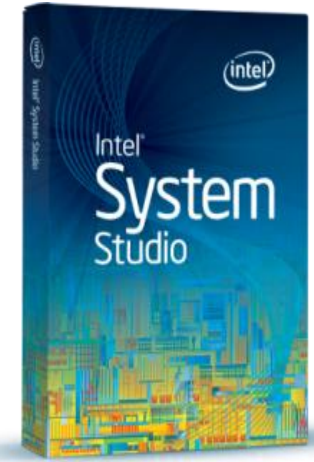
OS : Ubuntu 12.04 (64 bit)
Processor: Intel® Core™ i7-3770 @ 3.5GHz
Memory : 16GB
Processor Graphics SKU : GT2
Compiler version : Intel C++ Compiler 15.0 Update 1 Beta
HD Driver : 16.3.2.21305
Compiler option : -std=c++11 -xAVX

Optimization Notice

Intel® Software Development Tools

Advanced Performance

C++ and Fortran Compilers, MKL/IPP Libraries & Analysis Tools for Windows*, Linux* developers on IA based multi-core node



Deep System Insights for Embedded and Mobile Developers

Integrated software tool suite that provides deep system-wide insights to help:

- Accelerate Time-to-Market
- Strengthen System Reliability
- Boost power Efficiency and Performance

Optimization Notice

Synchronous Offload

- Annotate data parallel code section with #pragma offload target(gfx)
- Annotate functions invoked from the offloaded sections and global data with __declspec(target(gfx))
- Host thread waits for the offloaded code to finish execution
- Constraint - #pragma offload target(gfx) statement should be followed by a cilk_for loop
- Compiler automatically generates both host side as well as GFX code

Optimization Notice

Synchronous offload : Vector addition

```
void vector_add(float *c, float *a, float *b)
{
#pragma offload target(gfx) pin(a, b, c:length(ARRAYSIZE))
    cilk_for(int i = 0; i < ARRAYSIZE; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Optimization Notice

Quick look at Intel® Cilk™ Plus Array Notation

Original code:

```
for(int i = 0; i < N; i++)
    {
      for(int j = 0; j < N; j++)
      {
        c[i][j] = a[i][j] * b[j][i];
      }
    }
```

Array Notation version:

```
for(int i = 0; i < N; i++)
    {
      c[i][0:N] = a[i][0:N] * b[0:N][i];
    }
```

Optimization Notice

Synchronous offload – Matrix Multiplication Kernel

```
void matmul_tiled(float A[][K], float B[][N], float C[][N]) {
#pragma offload target(gfx) \
    pin(A:length(M)) pin(B:length(K)) pin(C:length(M))
    cilk_for (int m = 0; m < M; m += TILE_M) {                   // (a) iterate tile rows in the result matrix
        cilk_for (int n = 0; n < N; n += TILE_N) {               // (b) iterate tile columns in the result matrix
            // (c) Allocate current tiles for each matrix:
            float atile[TILE_M][TILE_K], btile[TILE_N], ctile[TILE_M][TILE_N];
            #pragma unroll
            ctile[:][:] = 0.0;                                  // initialize result tile

            for (int k = 0; k < K; k += TILE_K) {               // (d) calculate 'dot product' of the tiles
                #pragma unroll
                atile[:][:] = A[m:TILE_M][k:TILE_K];            // (e) cache atile in registers;
                #pragma unroll
                for (int tk = 0; tk < TILE_K; tk++) {           // (f) multiply the tiles
                    btile[:] = B[k+tk][n:TILE_N];               // (g) cache a row of matrix B tile
                    #pragma unroll
                    for (int tm = 0; tm < TILE_M; tm++) { // (h) do the multiply-add (MAD)
                        ctile[tm][:] += atile[tm][tk] * btile[:];
                    }
                }
            }
            #pragma unroll
            C[m:TILE_M][n:TILE_N] = ctile[:][:];  // (i) write the calculated tile to back memory
        }
    }
}
```

Optimization Notice

Key Points

- Tiles are declared as local arrays; they are small enough (default threshold is 3K) and their addresses do not 'escape', so the compiler will allocate them on registers.
- #pragma unroll for direct GRF addressing instead of inefficient indirect register addressing.
- Matrices are pinned – no data copying. The length clause is in elements – e.g. A is an M-element array of float arrays of length K.
- cilk_for is used to calculate 2D tiles in parallel
- btile could be 2D (btile[TILE_K][TILE_N]), but the higher dimension is reduced as it gives no extra performance but takes extra register space
- compiler will generate efficient series of octal word reads to fill the atile

Optimization Notice

Asynchronous Offload Support

- An API based offload solution
- By annotating functions with __declspec target(gfx_kernel)
- Above annotation creates the named kernel functions (Kernel entry points)
- Non-blocking API calls from CPU until the first explicit wait is specified.
- GFX kernels are enqueued into an in-order gpgpu queue
- Explicit control over data transfer, data decoupled from Kernel, data persistence across multiple kernel executions.
- Compiler just generates the GFX code.
- User to explicitly program the host version of offload section

Optimization Notice

Asynchronous offload – Vector addition

Host Code

```
float *a = new float[TOTALSIZE];
float *b = new float[TOTALSIZE];
float *c = new float[TOTALSIZE];
float *d = new float[TOTALSIZE];

a[0:TOTALSIZE] = 1;
b[0:TOTALSIZE] = 1;
c[0:TOTALSIZE] = 0;
d[0:TOTALSIZE] = 0;

_GFX_share(a, sizeof(float)*TOTALSIZE);
_GFX_share(b, sizeof(float)*TOTALSIZE);
_GFX_share(c, sizeof(float)*TOTALSIZE);
_GFX_share(d, sizeof(float)*TOTALSIZE);

_GFX_enqueue("vec_add", c, a, b, TOTALSIZE); // Non-blocking offload
_GFX_enqueue("vec_add", d, c, a, TOTALSIZE); // Place next kernel in
                                             // in-order queue
_GFX_wait();                                 // wait for all tasks

_GFX_unshare(a);
_GFX_unshare(b);
_GFX_unshare(c);
_GFX_unshare(d);
```

GPU Code

```
__declspec(target(gfx_kernel))
void vec_add(float *res, float *a, float *b, int size){
    cilk_for (int i = 0; i < size; i++)
    {
        res[i] = a[i] + b[i];
    }
    return;
}
```

Optimization Notice

Tuning tips for targeting GFX

- Collapse the nested loops by annotating with cilk_for. Will increase the iteration space. More h/w threads can be put into action.
- Pragma simd or Intel® Cilk™ Plus Array notation can be used to explicitly vectorize the offloaded code.
- Use __restrict__ keyword and __assume_aligned() to avoid compiler creating multiple code paths.
- Use pin clause to avoid the data copy overhead from DRAM to GPU memory. Enables data to be shared between CPU and GPU
- Consider using 4-byte elements rather than 1,2-bytes because gather/scatter operations of 4-byte elements are quite efficient but for 1,2-bytes elements they are much slower.
- Each local variable should be less than 3KB.

Optimization Notice

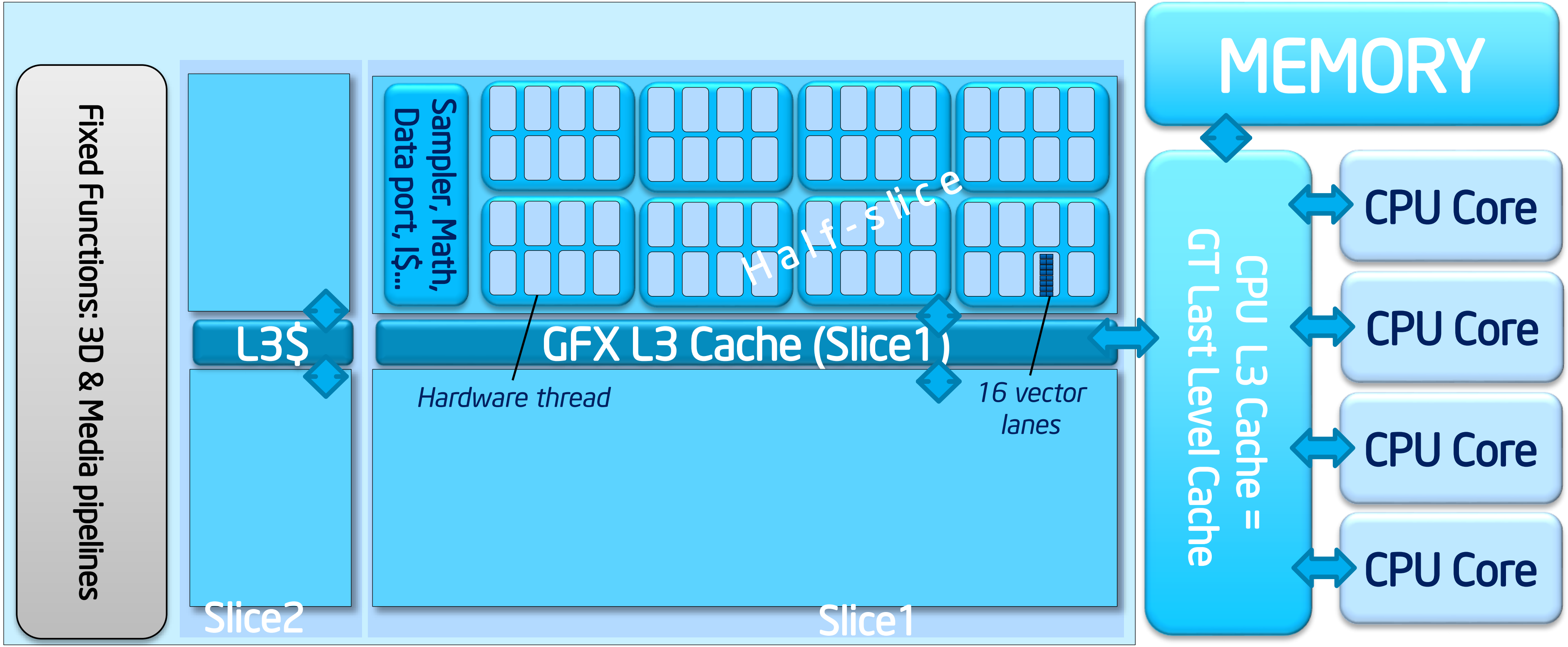
Tuning tips for targeting GFX contd…

- Considering SOA design over AOS for your data structures to avoid scatter/gather instructions.
- Strongest feature of GPU is 4KB private memory per h/w thread. All the local variables of the loop qualify for the register allocation. If the local variables cumulatively go beyond 4KB, the rest of the data needs to be accessed from much slower stack.
- For int buf[2048] allocated on GRF, for(i=0,2048) {… buf[i] … } will be an indexed register access. To enable direct register addressing, consider unrolling the loop. Excessive unrolling might lead to code bloat up and kernel size exceeding 250KB.
- JIT compiler might still spill some registers variables to memory impacting performance. Caching in local arrays should be done for 'hot' data.

Optimization Notice

GFX architecture (3rd generation Intel® Core™ Processors)

GFX building block hierarchy: Slice/Half slice/EU/thread/vector lane



Optimization Notice

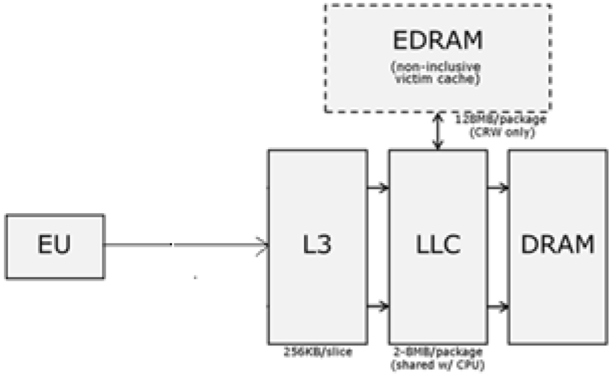
GFX architecture highlights (3rd generation Intel® Core™ Processors

- GFX is basically Fixed Functions + EUs + Caches
- EU is a general purpose highly parallel processor:
  - 6/8 H/W threads per EU
    - each thread has 4K register space (GRF – General Register File)
    - 16 vector lanes, 8 ops/cycle (16-wide ops take 2 cycles)
- Architecture Register File (Instruction Pointer, Address register etc).
- Memory model is covered in next slide.

Optimization Notice

Memory model – Cache hierarchy (3rd and 4th generation Intel® Core™ Processors

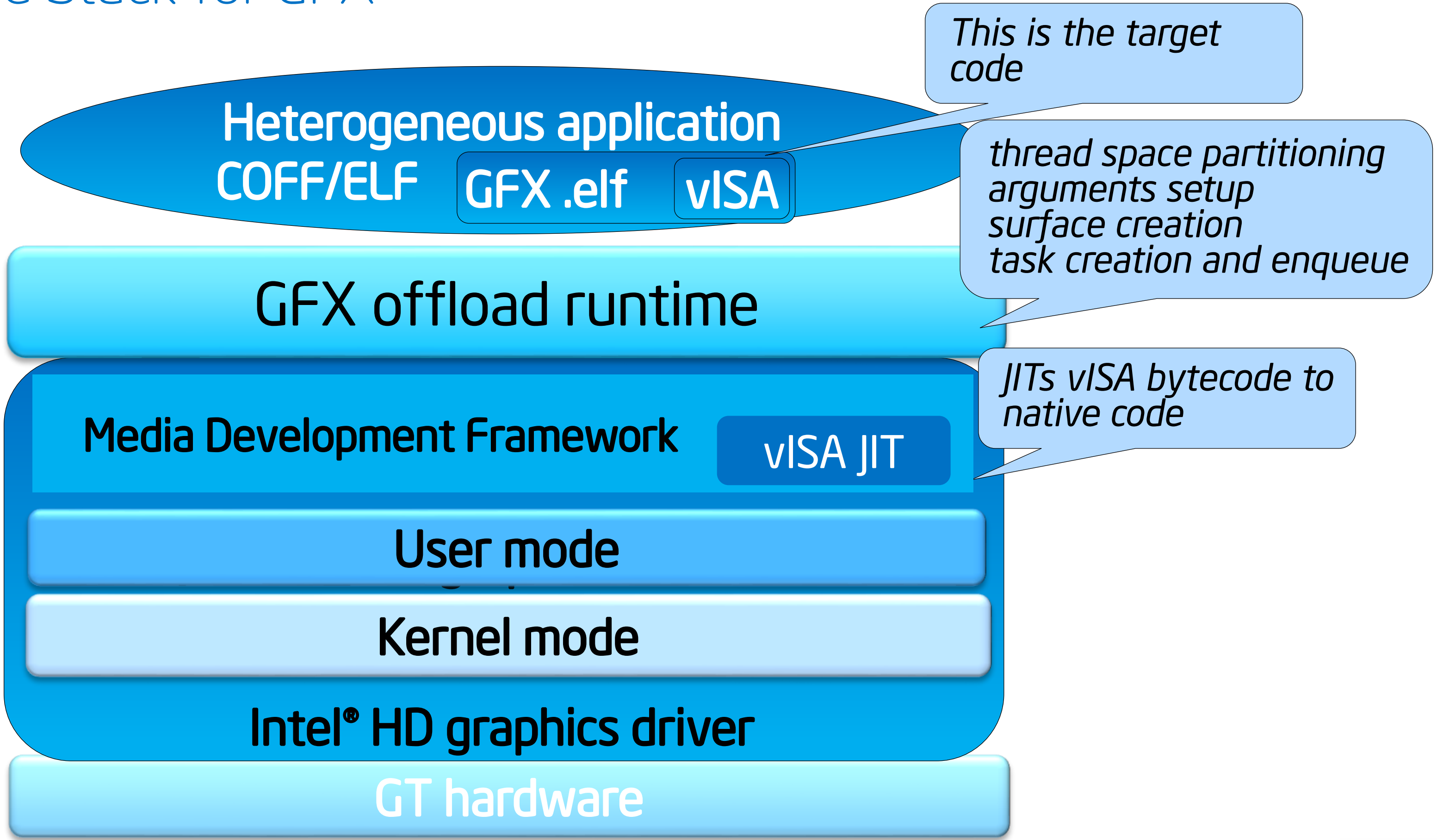
Cache hierarchy



- CPU L3 cache is shared between CPU and GPU. CPU L3 cache ⇔ GPU last level cache (LLC)
- Processor graphics comes with an L3 cache which is shared among all EUs in one slice.
- EDRAM – Embedded DRAM only applicable for GT3e.
- DRAM – System RAM (Global memory).
- Access to DRAM from GPU is cached in either LLC or L3 or eDRAM depending on the SKU of the processor graphics (Local memory).
- General Register File (GRF) of 4K per h/w thread (Register).
- No L1/L2 cache for GPGPU compute.

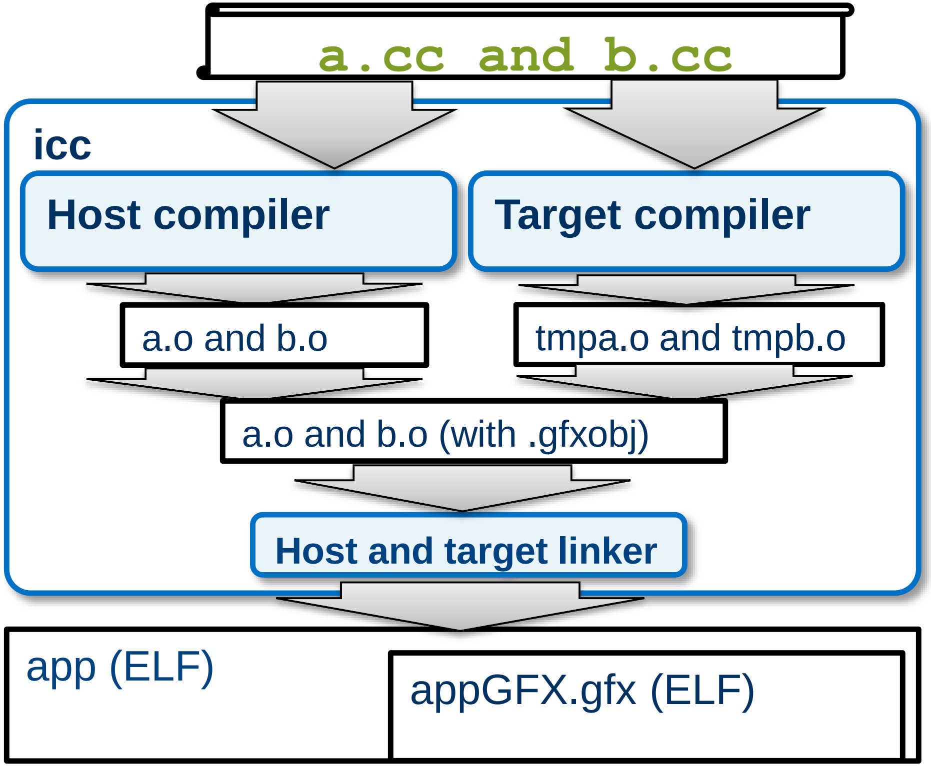
Optimization Notice

Software Stack for GFX



Optimization Notice

Intel® Graphics Technology compiler workflow



Compile time:

- icc –c a.cc b.cc

⇒ a.o and b.o

⇒ a.o and b.o has target object code as .gfxobj section

⇒ Temp target objects deleted

- icc a.o b.o –o app

⇒ Executable has target executable embedded

⇒ The target executable is extracted from fat executable using offload_extract tool (shipped with the compiler)

Optimization Notice

Gfxobj section

```
$ objdump -h a.o
app:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
..
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
 11 .gfxobj       00008f20  0000000000000000  0000000000000000  00001488  2**0
                  CONTENTS, READONLY
```

Optimization Notice

Hardware support

- 3rd and 4th generation Intel® Core™ Processors. These processors comes with either Intel® HD Graphics, Intel® Iris™ Graphics or Intel® Iris™ Pro Graphics.
- Intel® Pentium® Processors with processor model numbers 20xx and 3xxx.
- Intel® Celeron® Processors with processor model numbers 10xx and 29xx.
- For more information on the processors, please refer to http://ark.intel.com

Optimization Notice

OS platforms supported

Operating systems:

- Windows* 32/64 bit. On Windows* 7 (DX9) requires an active display, batch jobs are not supported. Above restriction is relaxed in Windows* 8 and Windows Server 2012*
- Linux* 64 bit
  - Ubuntu 12.04 (Linux kernel numbers: 3.2.0-41 for 3rd generation Intel® Core™ Processors and 3.8.0-23 for 4th generation Intel® Core™ Processors)
  - SLES11 SP3 (Linux kernel numbers: 3.0.76-11 for both 3rd and 4th generation Intel® Core™ Processors)
- No OS X* and Android* support as of now.

Optimization Notice

Limitations

- Main language restrictions
  - No exceptions, RTTI, longjmp/setjmp, VLA, variable parameter list, indirect control flow (virtual functions, function pointers, indirect calls and jumps)
  - No shared virtual memory
  - No pointer or reference typed globals
  - No OpenMP* or Intel® Cilk™ Plus tasking
- Runtime limitations
  - No ANSI C runtime library except math SVML library.
  - Inefficient 64-bit float and integer (due to HW limitations)
- No debugger support for GFX.

Optimization Notice

Call to Action

- Haven’t registered for Beta program yet? Please visit https://software.intel.com/en-us/articles/intel-software-development-tools-2015-beta for the same.
- Download and Install Intel® C++ Compiler 15.0 Update 1 Beta from http://registrationcenter.intel.com
- Downloa
  - For
  - For … nter.intel.com
- Binutils … com/en-us/artic
- GFX sa … cuting the samples. On Windows: gfx_samples.zip, On Linux: gfx_samples.tar.gz

Please evaluate both Synchronous and Asynchronous offload paradigms and provide your valuable feedback on the following:

1. If this programming paradigm fits your business needs.
2. Linux OS flavors and versions on which you wish to have this offload feature.

Optimization Notice

References

- GFX H/W specs documentation - https://01.org/linuxgraphics/documentation
- Intel® C++ Compiler 15.0 Beta User's Guide documentation
- Intel® Cilk™ Plus Webpage – http://cilkplus.org
- Vectorization Essentials using Intel® Cilk™ Plus - https://software.intel.com/en-us/articles/vectorization-essentials

Optimization Notice

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Comparison to OpenCL

OpenCL	GFX offload
Detecting platform and devices in each platform using clGetPlatformIDs() and clGetDevicesIDs() APIs. Supports this feature because OpenCL support in different GPUs shipped by different vendors.	Detection of integrated GPU is done by Intel® Graphics Technology offload runtime automatically.
The program to be run on GPU needs to be put in separate compilation unit (.cl) and build explicitly for target GPU from application host code using clCreateProgramWithSource() and clBuildProgram() APIs.	As long as the source code is annotated right with previously mentioned compiler hints, the existing code is ready to be run on integrated GPU.
Explicitly a command queue for the GPU using clCreateCommandQueue() API.	The command queue is created by the GFX offload runtime.
Kernels have to created from the program build for target using clCreateKernel() API at runtime.	The kernels are automatically created in both Synchronous offload and Asynchronous offload during the build process and not during the runtime.
Kernel invocation is not intuitive rather clEnqueueNDRangeKernel() API is used with the kernel name, number of work groups and number of work items/work group as parameters.	No code change required at the call site. Handled by GFX offload runtime.
Kernel function arguments are passed using clSetKernelArg() API	No code change required at the call site. Handled by GFX offload runtime.
Result from GPU after kernel computation is copied on to host memory using clEnqueueReadBuffer() API.	Synchronous offload – Copy of values are handled by the GFX runtime Asynchronous offload – Explicit copy from GPU to host required.
Explicitly release all kernel objects, memory objects, command queue, program and context object using clReleaseKernel(), clReleaseMemObject(), clReleaseCommandQueue(), clReleaseProgram() and clReleaseContext() APIs.	The release of task objects and buffer objects are done by the GFX runtime automatically.
Portable solution. Works with any GPU.	Non-portable solution. Works only for integrated GPU.

Optimization Notice

Generation and Jitting of vISA

- vISA generation is done by the compiler.
- Jitter translates vISA to GEN native ISA. Jitters are backward compatible to support previous generations of vISA.

Compiler	Supports 2nd generation Intel® Core™ Processors	Supports 3rd generation Intel® Core™ Processors	Supports 4th generation Intel® Core™ Processors
Intel® C++ Compiler 14.0 for GT	Y	Y	Y
Intel® C++ Compiler 15.0 Beta for GT	N	Y	Y

MDF runtime version	Supports 2nd generation Intel® Core™ Processors	Supports 3rd generation Intel® Core™ Processors	Supports 4th generation Intel® Core™ Processors
2.4	Y	Y	Y
3.0	N	Y	Y

Optimization Notice

Intrinsics for GFX

API	Description
_gfx_read_2d	Reads a rectangular area into memory pointed to by dst from a 2D image identified by img. The resulting area is laid out in dst memory linearly row by row.
_gfx_write_2d	Writes a rectangular area from memory pointed to by src to a 2D image identified by img.
_gfx_atomic_write_i32	Do atomic operations (enum: __GfxAtomicOpType) on previous values of the destination location using two source values. Used for serializing the operation on the same memory location across iterations for the vector loop. Returns previous value of the memory location.
_gfx_add_i8/i16/i32/f32/f64	Adding the two source inputs
_gfx_sub_i8/i16/i32/f32/f64	Subtraction using the two source inputs
_gfx_mullo_i8/i16/i32	Multiplication for char, short and int data types
_gfx_mul_f32/f64	Multiplication for float and double data types

Optimization Notice

Intrinsics for GFX (Contd…)

API	Description
_gfx_slli_i8/i16/i32	Left shift the operands (either in saturation mode or

Optimization Notice

EU Features

- SIMD instructions
- Instruction level variable-width SIMD execution
- Instruction compaction
- Conditional SIMD execution via destination mask, predication and execution mask
- Instruction can be executed in a SIMD pipeline consuming multiple clock cycles
- Region-based register addressing
- Direct or indirect (indexed) register addressing
- Instruction streams and Instruction cache are read only
- 3rd generation Intel® Core™ Processors has Gen7 generation of EU which allows dual-issue [Dual-Issue limited to most popular instructions]. Future generations supports more instructions with dual-issue.

Optimization Notice

GFX Instruction syntax

- (pred) inst cmod sat (exec_size) dst src0 src1 { inst_opt, … }
  - pred – predication
  - inst – instruction
  - cmod – conditional modifier
  - sat – saturation (clamping of the result to the nearest value)
  - exec_size – Max SIMD width specified for that instruction
  - dst – Specified as “rm.n<HorzStride>:type”
  - src0,src1 – Specified as “rm.n<VertStride;Width,HorzStride>:type”

The following example assembly language instruction adds two packed 16-element single-precision Float arrays in r4/r5 and r2/r3 writing results to r0/r1, only on those channels enabled by the predicate in f0.0 along with any other applicable masks.

```
(f0.0) add (16) r0.0<1>:f r2.0<8;8,1>:f r4.0<8;8,1>:f
```

Optimization Notice

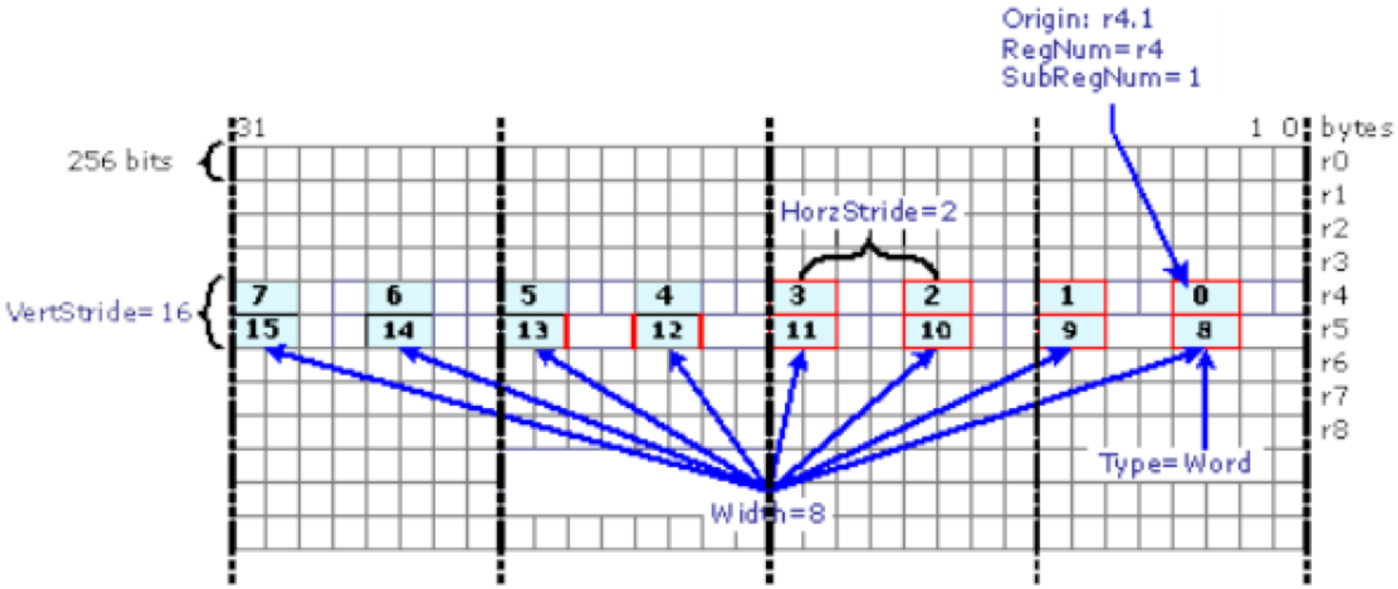
GFX Numeric Data types

- Integer data types:
  - 8-bit Unsigned/signed integer or byte (UB/B)
  - 16-bit unsigned/signed integer or word (UW/W)
  - 32-bit unsigned/signed integer or doubleword (UD/D)
  - Packed Unsigned Half-Byte Integer Vector, 8 x 4-bit unsigned integer (UV)
  - Packed Signed Half-Byte Integer Vector, 8 x 4-bit signed integer (V)
- Floating Point data types:
  - 32-bit single precision floating point number (F)
  - 64-bit double precision floating point number (DF)
  - Packed restricted float vector 8 x 4-bit restricted precision floating point number (VF)

Optimization Notice

Register region and Region Parameters

An example of a register region (*r4.1<16;8,2>:w*) with 16 elements



RefFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type=r4.1<16;8,2>:w

Optimization Notice

Categorization of Intel® HD Graphics

Graphics Level	PC Segment	Server Segment
GT3e	Intel® Iris™ Pro Graphics 5200	NA
GT3 (28W)	Intel® Iris™ Graphics 5100	NA
GT3 (15W)	Intel® HD Graphics 5000	NA
GT2	Intel® HD Graphics 4600/4400/4200	Intel® HD Graphics P4700/P4600
GT1	Intel® HD Graphics 2500	NA

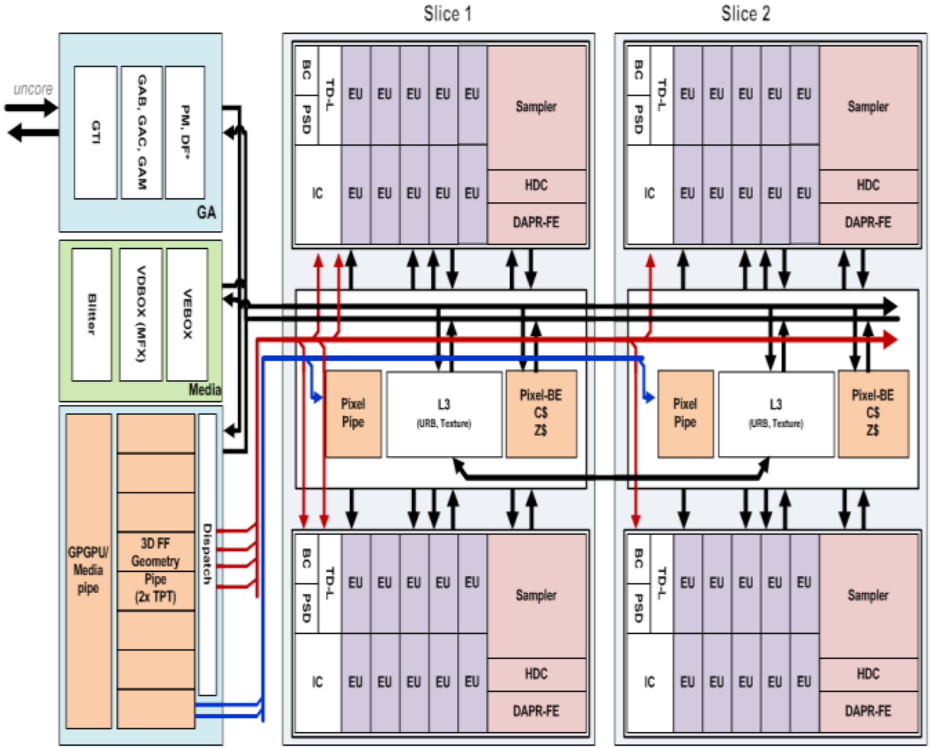
Optimization Notice

Comparison of 3rd generation processor graphics and 4th generation processor graphics

	3rd generation Intel® Core™ Processor	4th generation Intel® Core™ Processor		
	Intel® HD Graphics 4000	Intel® HD Graphics	Intel® HD Graphics 4200/4400/4600	Intel® Iris™ Pro Graphics 5200, Intel® Iris™ Graphics 5100, Intel® HD Graphics 5000
APIs	DirectX* 11.0 DirectX Shader Model 5.0 OpenGL* 4.0 OpenCL* 1.1	DirectX* 11.1 DirectX Shader Model 5.0 OpenGL* 4.2 OpenCL* 1.2		
Execution Units (EUs)	16 EUs	10 EUs	20 EUs	40 EUs
Floating Point ops per clock	256	160	320	640

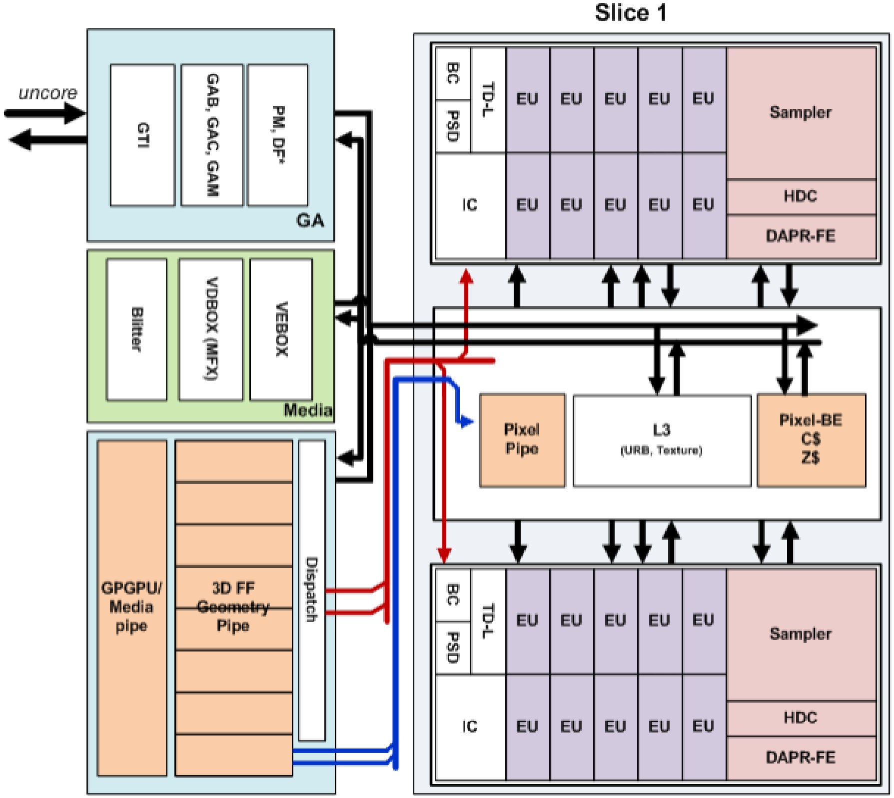
Optimization Notice

GT3 Top Level Block Diagram (4th generation Intel® Core™ Processors)



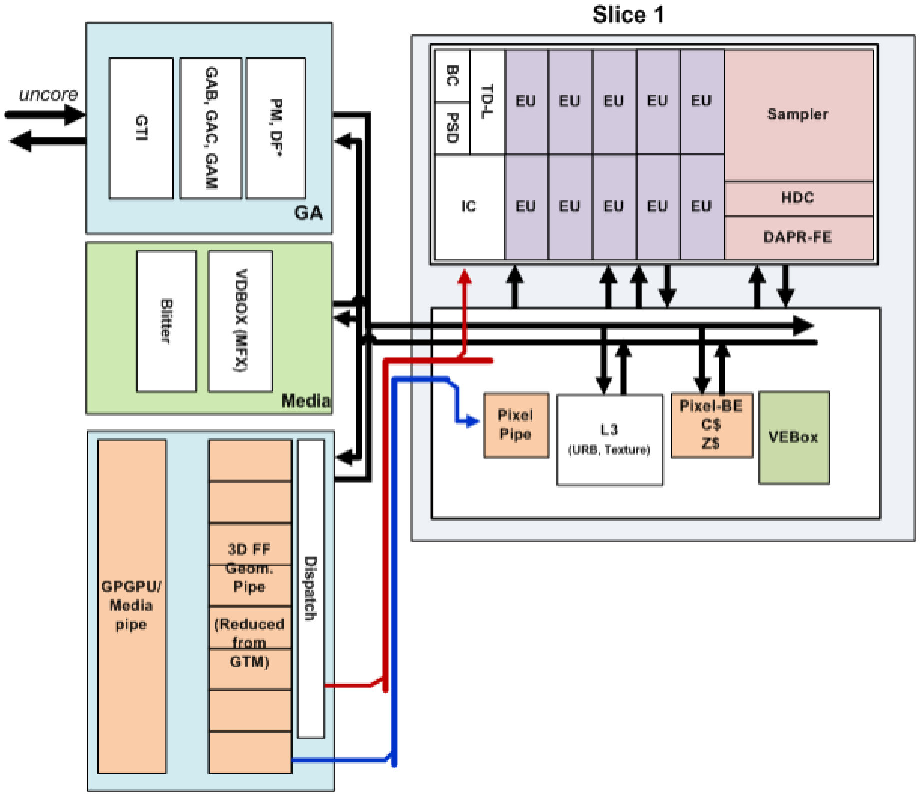
- 5 EUs Per row
- 2 rows per subslice
- 2 subslices per slice
- 2 slices (40 EUs total) in GT3
- 7 Threads Per EU
  - 280 threads in GT3
- 128 Registers per thread
  - 4KB per thread!
  - 1120KB in regfile in GT3
- 32K IC in each ROW (5EUs)
- 256KB data cache per slice (L3 only)

GT2 Top Level Block Diagram (4th generation Intel® Core™ Processors)



Optimization Notice

GT1 Top Level Block Diagram (4th generation Intel® Core™ Processors)



Optimization Notice

Asynchronous offload – Enqueue multiple kernels

```
__declspec(target(gfx_kernel))
void myKernel2(__gfx_surface_index si2d, int height, int width) {
    _cilk_for(int r = 0; r < height; r++){
        float tile[TILE_HEIGHT][TILE_WIDTH];
        for (int c = 0; c < width; c+=TILE_WIDTH){
            _gfx_read_2d(si2d, c*sizeof(float), r, tile, 32, 8);
            tile[:][:] += 20;
            _gfx_write_2d(si2d, c*sizeof(float), r, tile, 32, 8);
        }
    }
};
...
float * arr2dUnaligned = new float[h * w];
...
{
    GfxImage2D<float> i2d(arr2dUnaligned, h, w); // New API to represent 2D memory
    _GFX_enqueue("myKernel2", i2d, h, w);        // 2D image reuse between kernels.
    _GFX_enqueue("myKernel2", i2d, h, w);
    _GFX_wait();                                 // Blocks until all GPU tasks are done
    // i2d destructor is called here:
    // - writes data back to CPU
    // - sees that reference count is zero and
    //   destroys the underlying surface also
}
```

Optimization Notice

Hardware supported

- Hardware support

3rd/4th generation Intel® Core™ Processors	Intel® Pentium® Processors	Intel® Celeron® Processors
Intel® HD Graphics 2500 Intel® HD Graphics 4000	Processor Model Numbers: 20xx	Processor Model Numbers: 10xx
Intel® HD Graphics 4200 Intel® HD Graphics 4400 Intel® HD Graphics 4600 Intel® HD Graphics 5000 Intel® HD Graphics 5100 Intel® HD Graphics 5200	Processor Model Numbers: 3xxx	Processor Model Numbers: 29xx

- For more information on hardware, please visit: http://ark.intel.com

Optimization Notice

GFX Object Lifetime management

- The runtime shipped with HD driver maintains task, buffer and image objects during the program execution.
- Task object created when the kernel is enqueued and stays alive still successful completion of _GFX_wait().
- Buffer and Image objects are both reference managed. Runtime tracks the reference to these managed objects from the user code and from enqueued kernels.
- The buffer and image objects are destroyed by the runtime when reference to this object or reference is 0.
- Buffers are created lazily when kernel is enqueued to avoid redundant reallocations for many _GFX_share calls.

Optimization Notice

Comparison of GT3, GT2 and GT1 on 4th generation Intel® Core™ Processors

Description	GT3	GT2	GT1
Slice count	2	1	1
Subslice count	4	2	1
EUs (total)	40	20	10
Threads (total)	280	140	70
Threads/EU	7	7	7
L3 cache size	1024KB	512KB	256KB

Optimization Notice