Profiling Heterogeneous Computing Performance with VTune Profiler

Vladimir Tsymbal, vladimir.tsymbal@intel.com (presenter)
Alexandr Kurylev, alexandr.kurylev@intel.com
Contents

- Intel GPU and Microarchitecture
- GPU Development Environment and Runtimes
- CPU or GPU bound? VTune Offload Analysis
- GPU Task Efficiency
- Computing Tasks and Data Transfer
- Applications to offload to GPU. SYCL/DPC++, OpenCL, OpenMP
- In-kernel Analysis
- Memory Stalls in GPU Microarchitecture
- GPU Instructions Count
- Basic Blocks Latency and Memory Latency
- Platform Analysis
Intel® Gen9 HD Graphics

- Embedded to Coffee Lake SoC and newer
- Up to 48EUs x 7thr, up to 883 GFLOPS (SP)
- 2 SIMD-4 FPUs of 32-bit FP or INT data

Intel® Iris® Xe MAX discrete GPU

- 6 DSS x 16EUs (96 EUs x 7thr).
- VRAM 68 GB/s
- PCIe3x16 card, 2456 GFLOPS (SP)
GPU Development Environment and Runtimes

Several high-level languages for Media and GPGPU programming

- OpenCL™ Technology via Intel® Media SDK
- SYCL/Data Parallel C++ direct programming
- OpenMP offload to GPU
- Performance Libraries

Set of Intel Compilers based on LLVM technology

<table>
<thead>
<tr>
<th>Intel Compiler</th>
<th>Target</th>
<th>OpenMP Support</th>
<th>OpenMP Offload Support</th>
<th>Included in oneAPI Toolkit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® oneAPI DPC++/C++ Compiler dpcpp</td>
<td>CPU, GPU, FPGA*</td>
<td>Yes</td>
<td>Yes</td>
<td>Base</td>
</tr>
<tr>
<td>Intel® oneAPI DPC++/C++ Compiler icx</td>
<td>CPU, GPU*</td>
<td>Yes</td>
<td>Yes</td>
<td>Base</td>
</tr>
<tr>
<td>Intel® Fortran Compiler ifx</td>
<td>CPU, GPU*</td>
<td>Yes</td>
<td>Yes</td>
<td>HPC</td>
</tr>
</tbody>
</table>
Runtime Architecture

- Controlled via SYCL_BE env var:
  - PI_OPENCL
  - PI_LEVEL0

- Performance data comes from most of the levels

More info: tinyurl.com/dpcpp-pi
CPU or GPU bound? VTune Offload Analysis

All execution resources in focus
- Explore code execution on various CPU and GPU cores
- Correlate CPU and GPU activity
- Identify whether your application is GPU or CPU bound

Find kernels for further analysis
- Task level analysis
- Kernel efficiency
- Data transfer rates
GPU Task Efficiency

**Recommendations**

EU Array Stalled/Idle: 98.4%
GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots (preview) to understand how well your application runs on the specified hardware.

Execution % of Total Time: 0.1%
Execution time on the device is less than memory transfer time. Make sure your offload schema is optimal. Use Intel Advisor tool to get an insight into possible causes for inefficient offload. Learn more.

**Elapsed Time:** 60.013s
- GPU Utilization: 88.9%

**Hottest GPU Computing Tasks**
This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

<table>
<thead>
<tr>
<th>Computing Task</th>
<th>Total Time</th>
<th>Execution</th>
<th>% of Total Time</th>
<th>Instance Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrixMultiply1&lt; float, (unsigned long)2048&gt; (void, std::array&lt;std::array&lt;float, (unsigned long)2048&gt;, (unsigned long)2048&gt; const, std::array&lt;std::array&lt;float, (unsigned long)2048&gt;, (unsigned long)2048&gt; const, std::array&lt;float, (unsigned long)2048&gt;&amp;) (lambda(cl: sycl::handler&amp;)[1]::operator()) (cl: sycl) const=MatrixMultiply1 &amp; z.&amp;</td>
<td>51.943s</td>
<td>0.053s</td>
<td>0.1%</td>
<td>2,010</td>
</tr>
<tr>
<td>zeCommandListAppendBarrier</td>
<td>0.002s</td>
<td>0s</td>
<td>0.0%</td>
<td>0</td>
</tr>
</tbody>
</table>
**GPU Task Efficiency**

**Recommendations**

EU Array Stalled/Idle: 36.4%

GPU metrics detect some kernel issues. Use [GPU Compute/Media Hotspots](#) to understand how well your application runs on the specified hardware.

Execution % of Total Time: 3.1%

Execution time on the device is less than memory transfer time. Make sure your offload schema is optimal. Use [Intel Advisor](#) tool to get an insight into possible causes for inefficient offload. [Learn more](#)

**Elapsed Time**: 60.013s

**GPU Utilization**: 88.9%

**Hottest GPU Computing Tasks**

This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

<table>
<thead>
<tr>
<th>Computing Task</th>
<th>Total Time</th>
<th>Execution</th>
<th>% of Total Time</th>
<th>Instance Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrixMultiply1&lt;float, (unsigned long)2048&gt; void std::array&lt;std::array&lt;float, (unsigned long)2048&gt;, (unsigned long)2048&gt; const std::array&lt;std::array&lt;float, (unsigned long)2048&gt;, (unsigned long)2048&gt;&amp;, (lambda&lt;cl::sycl::handler&amp; h1,1::operator()<a href="">cl::sycl</a> const::MatrixMultiply1&gt;) zzeCommandListAppendBarrier</td>
<td>51.943 s</td>
<td>0.053 s</td>
<td>0.1%</td>
<td>2,010</td>
</tr>
</tbody>
</table>
GPU Task Efficiency

**Recommendations**
EU Array Stalled/Idle: 98.4%
GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots (preview) to understand how well your application runs on the specified hardware.

Execution % of Total Time: 0.1%
Execution time on the device is less than memory transfer time. Make sure your offload scheme is optimal. Use Intel Advisor tool to get an insight into possible causes for inefficient offload. Learn more

**Elapsed Time**: 60.013s
- GPU Utilization: 88.9%

**Hottest GPU Computing Tasks**
This section lists the most active computing tasks running on the GPU, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

<table>
<thead>
<tr>
<th>Computing Task</th>
<th>Total Time</th>
<th>Execution</th>
<th>% of Total Time</th>
<th>Instance Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrixMultiply1&lt;float, (unsigned long)2048&gt; (void, std::array&lt;st::array&lt;float, (unsigned long)2048&gt; &amp;), std::array&lt;float, (unsigned long)2048&gt; consi, std::array&lt;float, (unsigned long)2048&gt; &amp;); (lambda(0::sycl::handler&amp;0::operator()) (0::sycl) const::MatrixMultiply1) zzeCommandListAppendBarrier</td>
<td>51.943s</td>
<td>0.053s</td>
<td>0.1%</td>
<td>2,010</td>
</tr>
<tr>
<td>zzeCommandListAppendBarrier</td>
<td>0.002s</td>
<td>0s</td>
<td>0.0%</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing Tasks and Data Transfer

![Image of Intel VTune Profiler interface showing GPU Offload for Graphics]

### Computing Task and Data Transfer Analysis

- **Matrix Multiply**
  - **Execution Time:** 51.943s
  - **Transfer Size:** 94 GB
  - **Work Size:** 2048 x 512

- **Device-to-Device Transfer**
  - **Transfer Size:** 0 B

- **Device-to-Host Transfer**
  - **Transfer Size:** 0 B

- **Host-to-Device Transfer**
  - **Transfer Size:** 0 B

---

**Thread Analysis**

- **Matrix Multiply (TID: 13204...)**
  - **Execution Time:** 8730ms
  - **GPU Execution Units Utilization**
    - **Active**
    - **Idle**

- **Matrix Multiply (TID: 13205...)**
  - **Execution Time:** 8740ms
  - **GPU Execution Units Utilization**
    - **Active**
    - **Idle**

- **Matrix Multiply (TID: 13205...)**
  - **Execution Time:** 8750ms
  - **GPU Execution Units Utilization**
    - **Active**
    - **Idle**

---

**Additional Analysis Options**

- **CPU Time**
- **Spin and Overhead**
- **Clocktick Sample**
- **User Tasks**
- **GPU Computing Units**
- **EU Arrays**
- **GPU Execution Units**
  - **Active**
  - **Idle**
Computing Tasks and Data Transfer
Computing Tasks and Data Transfer
Computing Tasks decomposition

- **User Tasks**
  - Start: 8751.718ms
  - Duration: 1.821ms
  - Task Type: `zeMemAllocDevice`

- **User Tasks**
  - Start: 8753.556ms
  - Duration: 5.737ms
  - Task Type: `zeCommandListAppendMemoryCopy`

- **User Tasks**
  - Start: 8759.342ms
  - Duration: 16.927usec
  - Task Type: `zeCommandQueueExecuteCommandLists`

- **User Tasks**
  - Start: 8759.499ms
  - Duration: 4.379ms
  - Task Type: `zeEventHostSynchronize`
Applications to offload to GPU. SYCL/DPC++

DPC++ “Hello World”: Vector Addition

```cpp
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };
        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Applications to offload to GPU. OpenCL

```c
__kernel void vector_add(__global const float *x,
                        __global const float *y,
                        __global float *restrict z)
{
    // get index of the work item
    int index = get_global_id(0);
    // add the vector elements
    z[index] = x[index] + y[index];
}
```

**Top Tasks**

This section lists the most active tasks in your application.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Task Time</th>
<th>Task Count</th>
<th>Average Task Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>clBuildProgram</td>
<td>0.237s</td>
<td>1</td>
<td>0.237s</td>
</tr>
<tr>
<td>clCreateBuffer</td>
<td>0.118s</td>
<td>3</td>
<td>0.039s</td>
</tr>
<tr>
<td>clCreateKernel</td>
<td>0.016s</td>
<td>1</td>
<td>0.016s</td>
</tr>
<tr>
<td>clCreateContext</td>
<td>0.000s</td>
<td>1</td>
<td>0.000s</td>
</tr>
<tr>
<td>clCreateCommandQueueWithProperties</td>
<td>0.000s</td>
<td>1</td>
<td>0.000s</td>
</tr>
</tbody>
</table>
Applications to offload to GPU. OpenCL

```c
void run() {

    cl_int status;
    const double start_time = getCurrentTimestamp();
    // Launch the problem for each device.
    scoped_array<cl_event> kernel_event(num_devices);
    scoped_array<cl_event> finish_event(num_devices);
    for(unsigned i = 0; i < num_devices; ++i) {
        // for the host-to-device transfer.
        cl_event write_event[2];
        status = clEnqueueWriteBuffer(queue[i], input_a_buf[i], CL_FALSE, 0, n_per_device[i] * sizeof(float), input_a[i], 0, NULL, &write_event[0]);
        checkError(status, "Failed to transfer input A");
        status = clEnqueueWriteBuffer(queue[i], input_b_buf[i], CL_FALSE, 0, n_per_device[i] * sizeof(float), input_b[i], 0, NULL, &write_event[1]);
        checkError(status, "Failed to transfer input B");
        // Set kernel arguments.
        unsigned argi = 0;
        status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_a_buf[i]);
        checkError(status, "Failed to set argument %d", argi - 1);
        status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_b_buf[i]);
        checkError(status, "Failed to set argument %d", argi - 1);
        status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &output_buf[i]);
        checkError(status, "Failed to set argument %d", argi - 1);
        const size_t global_work_size = n_per_device[i];
        status = clEnqueueNDRangeKernel(queue[i], kernel[i], 1, NULL, &global_work_size, NULL, 2, write_event, &kernel_event[i]);
        checkError(status, "Failed to launch kernel");
        // Read the result. This is the final operation.
        status = clEnqueueReadBuffer(queue[i], output_buf[i], CL_FALSE, 0, n_per_device[i] * sizeof(float), output[i], 1, &kernel_event[i], &finish_event[i]);
        checkError(status, "Failed to read output");
        // Release local events.
        clReleaseEvent(write_event[0]);
        clReleaseEvent(write_event[1]);
        // Wait for all devices to finish.
        clWaitForEvents(num_devices, finish_event);
        // Release all events.
        for(unsigned i = 0; i < num_devices; ++i) {
            clReleaseEvent(kernel_event[i]);
        }
    }
}
```

Applications to offload to GPU. OpenMP offload

```c
void __attribute__((noinline)) MatrixMulOpenMpGpuOffloading() {
    int i, j, k;

    // Each element of matrix a is 1.
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++) a[i][j] = 1.0f;

    // Each column of b is the sequence 1,2,...,N
    for (i = 0; i < N; i++)
        for (j = 0; j < P; j++) b[i][j] = i + 1.0f;

    // c is initialized to zero.
    for (i = 0; i < M; i++)
        for (j = 0; j < P; j++) c[i][j] = 0.0f;

    // Parallelize on target device.
    #pragma omp target teams distribute parallel for map(to : a, b) \map(tofrom : c) thread_limit(128)
    {
        for (i = 0; i < M; i++)
            for (k = 0; k < N; k++)
                // Each element of the product is just the sum 1+2+...+n
                for (j = 0; j < P; j++)
                    c[i][j] += a[i][k] * b[k][j];
    }
}
```

[Link to the code on GitHub](https://github.com/oneapi-src/oneAPI-samples/blob/master/DirectProgramming/DPC%2B%2B/DenseLinearAlgebra/matrix_mul/src/matrix_mul_omp.cpp)
### In-kernel Analysis

#### GPU Compute/Media Hotspots

Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. Learn more.

- **Characterization**: Select the type of analysis. Options include:
  - **Overview**

**GPU sampling interval, ms**

- 1

**Analyze memory bandwidth**

- **On**

**Trace GPU programming APIs**

- **On**

---

#### GPU Adapter / Computing Task

<table>
<thead>
<tr>
<th>GPU Adapter / Computing Task</th>
<th>Work Size</th>
<th>Computing Task</th>
<th>Data Transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global</td>
<td>Local</td>
<td>Total Time</td>
</tr>
<tr>
<td>DG1 [Iris Xe MAX Graphics]</td>
<td>2048 x 2048</td>
<td>512 x 1</td>
<td>1.990s</td>
</tr>
<tr>
<td>matrix multiply2&lt;float, (unsigned long)2048&gt;(v)</td>
<td>zeCommandListAppendMemoryCopy</td>
<td>0.017s</td>
<td>0.006s</td>
</tr>
<tr>
<td></td>
<td>zeCommandListAppendMemoryCopyRegion</td>
<td>0.160s</td>
<td>0.160s</td>
</tr>
<tr>
<td></td>
<td>zeCommandListAppendBarrier</td>
<td>0.000s</td>
<td>0.000s</td>
</tr>
</tbody>
</table>
### GPU Instructions Count

**Instructions decomposition for a Computing Task and underlaying functions**

<table>
<thead>
<tr>
<th>Computing Task / Function / Call Stack</th>
<th>GPU Instructions Executed by Instruction Type</th>
<th>SIMD Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control Flow</td>
<td>Send</td>
</tr>
<tr>
<td>matrixMultiply2&lt;float, (unsigned long)20</td>
<td>33,554,432</td>
<td>1,611,005,952</td>
</tr>
<tr>
<td>matrixMultiply2&lt;float, (unsigned long)20</td>
<td>0</td>
<td>131,072</td>
</tr>
<tr>
<td>__spirv_GlobalInvocationId_y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;event, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20))</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>matrixMultiply2&lt;float, (unsigned long)20</td>
<td>33,554,432</td>
<td>1,610,874,880</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::accessor&lt;float, (unsigned long)20)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
GPU Instructions Count

Instructions decomposition for a source line
# Basic Blocks Latency

<table>
<thead>
<tr>
<th>Computing Task / Function / Call Stack</th>
<th>Work Size</th>
<th>Computing Task</th>
<th>Data Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global ▼</td>
<td>Local ▼ Total Time</td>
<td>Average Time</td>
</tr>
<tr>
<td>matrixMultiply2&lt; float, (unsigned long)2048&gt;(void)</td>
<td>2048 x 2048</td>
<td>512 x 1</td>
<td>199.224ms</td>
</tr>
<tr>
<td>cl::sycl::accessor&lt; float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrixMultiply2&lt; float, (unsigned long)2048&gt;(void)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cl::sycl::accessor&lt; float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Define function calls that took most of GPU cycles
# Basic Blocks Latency

<table>
<thead>
<tr>
<th>Computing Task / Function / Call Stack</th>
<th>Work Size</th>
<th>Computing Task</th>
<th>Data Transfer</th>
<th>Estimated GPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global ▼ Local Total Time Average Time Instance ▼ SIMD Width SIMD Width SVM... Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ matrixMultiply2&lt;float, (unsigned long)2048&gt;(void)</td>
<td>2048 x 2048 512 x 1 199.224ms 199.224ms 1 32 32</td>
<td></td>
<td></td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>▼ matrixMultiply2&lt;float, (unsigned long)2048&gt;(s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td>14.2%</td>
</tr>
<tr>
<td></td>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td>20.1%</td>
</tr>
<tr>
<td></td>
<td>cl::sycl::accessor&lt;float, (int)2, (cl::sycl::access</td>
<td></td>
<td></td>
<td>0.7%</td>
</tr>
</tbody>
</table>

## Source Code Analysis

```cpp
145       auto rc = bc.template get_access<dpcpp::access::mode::discard_write>(hdr);;
146       hdr.paralle_for<class MatrixMultiply2>(matrixRange, [=](dpcpp::id<2> id)
148             { size_t i = id[0], j = id[1];
150             rc[i][j] = T();
152             for (size_t k = 0; k < w; k++)
153                 rc[i][j] += ra[i][k] * rb[k][j];
156         });
```

Estimated GPU Cycles:
- 145: 0.2%
- 147: 0.0%
- 148: 0.0%
- 150: 2.6%
- 154: 41.9%
Memory Latency

Latencies per individual instructions
Platform Analysis

Elapsed Time: 60.013s
GPU Utilization: 88.9%

Bandwidth Utilization Histogram
Bandwidth Domain:
Bandwidth Utilization

- PCIe Bandwidth, MB/sec
- DRAM, GB/sec

CPU Time
- Active
- Idle
- Stalled

GPU Computing Threads
- Computing Thread
- EU Threads Occupied

GPU Execution Units
- EU Arrays
  - Active
  - Idle
  - Stalled

GPU Utilization
- Render/GPGPU Compute
- CPU Time
- CPU Time

Inbound PCIe Bandwidth
- Inbound PCIe Bandwidth
  - Read
  - Write
  - Total, MB/sec
Quick References

Intel® VTune™ Profiler – Performance Profiler
- Product page – overview, features, FAQs...
- Training materials – Cookbooks, User Guide, Processor Tuning Guides
- Support Forum
- Online Service Center - Secure Priority Support
- What’s New?

Additional Analysis Tools
- Intel® Advisor – Design and optimize for efficient vectorization, threading, memory usage, and accelerator offload. Roofline and flow graph analysis.
- Intel® Inspector – memory and thread checker/ debugger
- Intel® Trace Analyzer and Collector - MPI Analyzer and Profiler

Additional Development Products
- Intel® Software Development Products
Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.