Jetson TK1

Benjamin Baumann
Department of Computer Engineering
University of Heidelberg
Germany, 68131 Mannheim
Email: Benjamin.Baumann@stud.uni-heidelberg.de

***Abstract*—In the paper *GPU Computing* by *J. D. Owens, M. Houston et al., 2008* the authors stated that *the graphics processing unit (GPU) has become an integral part of today's mainstream computing systems*[1]. Seven years later the GPU is going to become an integral part of embedded computing. The ongoing automation in industrial fields and automated driving made image processing a key workload for autonomous systems. So combining a CPU and GPU in one system on chip (SOC) is a useful step to reduce consumption and cost of these systems. The NVIDIA Tegra K1 is the first SOC with a CUDA capable general purpose GPU (GPGPU) and an energy efficient ARM CPU to fulfill these parallel workloads and high demands on efficient energy consumption and high performance.**
This paper gives an insight into the SOC and examines whether it can be used as a lower power high performance System. Furthermore, the chip is compared with an existing high performance system and the results are interpreted.

***Keywords*—CUDA, GPU, SOC, Embedded GPGPU, Tegra K1, parallel computing, UVM**

I. Introduction

Automation is a key part of todays industry. An increasing number of work is done by robot-like systems. Image processing is a major task for these systems so it is understandable that graphics cards are used. But GPGPUs have a high energy consumption and are too large for embedded systems.

In the beginning the GPU was only a fixed function processor. It was used to render the threedimensional graphics but not much in addition. Over the years the GPU evolved into a programmable processor with a extensive application programming interface (API). But not only the software side progressed also the hardware focused on adding programmability to the GPU.

Current GPU have an immense arithmetic capability and a high streaming memory bandwidth which is both greater than in a high-end CPU. Considering this, it's no surprise that throughput-oriented applications can benefit most from GPUs.

A GPU can execute thousands of threads concurrently because it consists of a large number of fine-grained parallel processors. To get most performance out of all these thousands of cores an advancement of the programming model and the programming tools is necessary. The GPU need to balance between low-level access to the hardware ton enable performance and high level programming languages and tools that allow programmer flexibility and productivity. NVIDIA is offering CUDA which is integrated in C/C++. CUDA provides easy programmability and low level access to the architecture. In Section II-B, CUDA is described in detail.

In the beginning using GPUs could best be described as an academic exchange [1]. Fast GPUs demonstrated an appreciable advantage also in real applications. Limited by CPU performance, the high performance computing community rapidly adopted GPUs and offloaded complex CPU tasks to the GPU which yields better overall performance. In 2013 19% of FLOPS (floating point per second) in the TOP500 list where achived by GPU systems [2].

II. GPU Computing

GPU computing is the heterogeneous computing of the CPU and GPU. The programmer has the choice between high throughput of parallel code, the GPU, and high single thread performance and low latency, the CPU. So the application has to be divided in an parallel section and serial section. In figure 1 it is shown how heterogeneous computing is used.

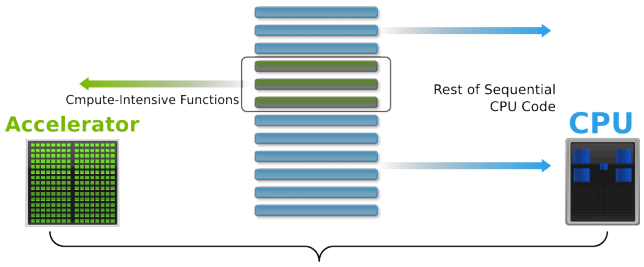

Fig. 1. Application Code and Heterogeneous Computing [3]

Over the years many applications have been ported to GPUs and this is an ongoing process. Only between 2010 and 2012 the number of applications has more than doubled. [2]

A. GPU Architecture

GPUs are throughput-oriented and have a relaxed latency compared to CPUs. To still have these high throughput GPUs can change threads in one clock cycle. In figure 2 can be seen that one streaming multiprocessor has 32 cores, this is called a thread warp. All these cores are instructed by the same scheduler. [4]

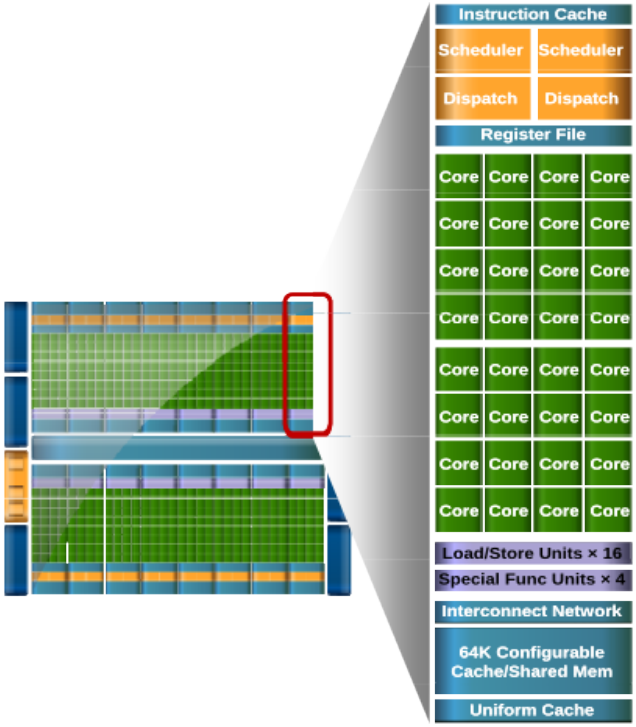

Fig. 2. Streaming Multiprozessor in Fermi architecture [5]

GPUs are build for different application demands than CPUs. They are build for large parallel computation requirements with priority on throughput rather than latency. So the architecture has progressed in a different direction than that of the CPU. [1]
To fulfill the needs of large parallel computation workloads the GPUs consist of up to thousands of cores.

B. CUDA Platform and Programming Model

CUDA is general purpose parallel computing platform and programming model. With CUDA the parallel compute engine in NVIDIA GPUs can solve many complex computational problems in a more efficient way than a CPU. The CUDA parallel programming model is designed to scale its parallelism to the increasing number of processor cores like three-dimensional graphics applications do. Developers can use C and C++ as a high-level programming language in the CUDA environment. C and C++ is used to maintain a low learning curve for programmers, who are used to use these programming languages. [6]

The programming model of CUDA consists of three key abstractions:

- hierarchy of thread groups
- shared memories
- barrier synchronization

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. [6]

The positive side-effect of this partition is, that the application gains automatic scalability. Each block of threads can be scheduled to any of the available streaming multiprocessor, see figure 2. Also it is possible to schedule the threads concurrently or sequentially, so the runtime system only needs to know the number of streaming multiprocessors and can optimize the scheduling for any given GPU.

```
// Allocate two N-vectors h_x and h_y
int size = N * sizeof(float);
float* h_x = (float*)malloc(size);
float* h_y = (float*)malloc(size);

// Initialize them...

// Allocate device memory
float* d_x; float* d_y;
cudaMalloc((void**)&d_x, size));
cudaMalloc((void**)&d_y, size));

// Copy host memory to device memory
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (N + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(N, 2.0, d_x, d_y);

// Copy result back from device memory to host memory
cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost);
```

Fig. 3. SAXPY: Host Code [5]

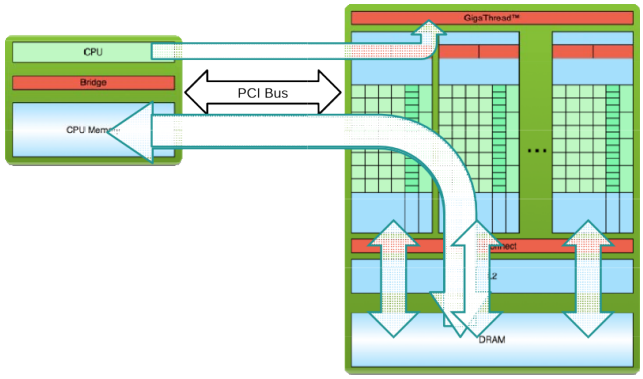

Fig. 4. Processing flow for GPU transfers [5]

In figure 3, CUDA C Code for Host machines can be seen. At first memory for the variables on the host machine is allocated. Then the same size is allocated on the machine with the CUDA instruction **cudaMalloc**. After the host variables have to be copied to the device memory with the **cudaMemcpy** function. Now the **saxpy_parallel** kernel is started. The values between the <<<... >>> tells the runtime how many blocks and how many threads per

block have to be started. After the computation of the **saxpy_parallel** kernel the result has to be copied back to the system. These two copy instructions are a major bottleneck of actual heterogeneous computing. In figure 4 can be seen that the data has to be move from the CPU memory over the PCI Bus to the device memory and back over the PCI memory to the CPU memory.

```
void saxpy_serial(int n, float a, float *x, float *y)
{
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n)  y[i] = a*x[i] + y[i];
}
```

Fig. 5. SAXPY serial and SAXPY parallel [5]

Figure 5 shows a the **saxpy_serial** code for the CPU and the **saxpy_parallel** code for the GPU kernel. The biggest difference between these to code snippets is that the parallel version has no for-loop. The serial processing of each element is now done parallel by many threads. The loop variable **i** is now a unique thread identifier and assigns one element to each thread. The **__global__** shows that the function **saxpy_parallel** can be called by the host, like in figure 3 and is processed by the device.

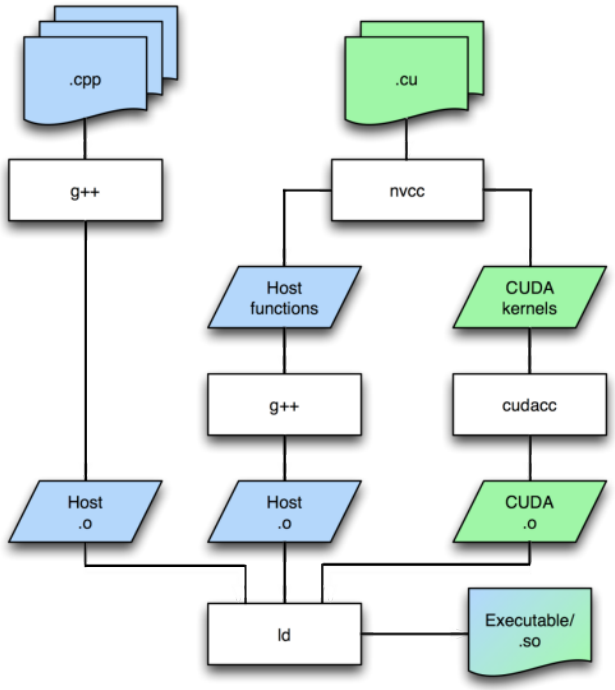

Fig. 6. Compilation on a Linux system [5]

Compiling the heterogeneous code is shown in figure 6. The command for compiling the **.cu** is **nvcc kernel.cu -o program.o**. In the figure is shown the **nvcc** divides the host functions from the CUDA kernels and compiles the CPU code with **g++** and the GPU with **cudacc**. After compiling is finished the host output and device output is combined to one executable.

III. Jetson TK1

The embedded system board Jetson TK1 was presented in March 2014 by NVIDIA. This board is based on the hybrid processor Tegra K1. The Tegra K1 is schon in figure 7. It consists of a quad-core ARM Cortex A15 CPU with a battery saver core and a NVIDIA Kepler Core with 192 computational cores.

Furthermore the board offers:

- 1 mini-PCIe slot
- 1 SD/MMC connector
- 1 HDMI port
- 1 USB 2.0 port
- 1 USB 3.0 port
- 1 RS232 serial port
- 1 GigE LAN
- 1 SATA data port
- SPI 4MByte boot flash

The board comes with a pre-installed Linux and the CUDA environment can be downloaded and installed on the system.

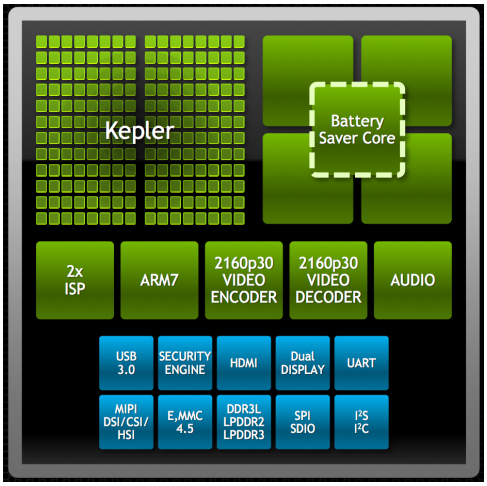

Fig. 7. Tegra K1 on Jetson TK1 [7]

Tegra K1

In figure 7 the Tegra K1 is shown. Compared to the GPU Tesla K20m from the benchmark in section IV the Kepler GPU in the Tegra K1 is pretty small. The K20m has 2496 CUDA cores and the Tegra K1 only 192 CUDA cores, in figure 8 the sizes of the different NVIDIA GPU CUDA cores is shown.

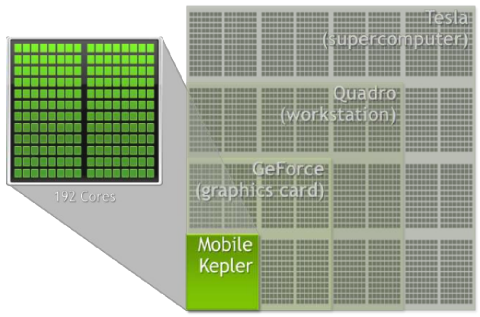

Fig. 8. Kepler Core on Tegra K1 [3]

Memory: Due to sharing the on chip memory with the CPU, there are no communication overheads between the CPU and the GPU, avoiding the major performance bottleneck found in heterogeneous systems with a discrete GPU. In figure 9 the difference of a discrete GPU system and the shared physical memory of the Jetson TK1 is shown. [8]

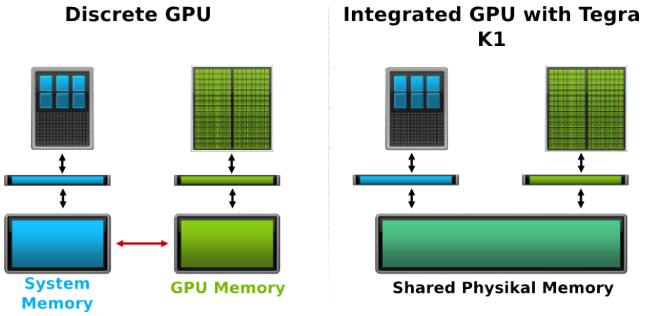

Fig. 9. Unified physical memory on Tegra K1 [3]

To use this advantage the programmer hast to use the **zero copy** option in the CUDA environment. With the **cudaHostAlloc** function the programmer can allocate memory on the host which is accessible from the GPU. With **cudaHostGetDevicePointer** a pointer for the device to the memory can be generated. Now the GPU can directly access the host memory with this pointer and there is no need for a **cudaMemcpy**. A huge advantage of the Tegra K1 is that the CPU and the GPU share the same memory, see figure 10. The GPU can still use the cache benefits. If the **zero copy** option is used on a discrete GPU all the caching benefits get lost.

IV. Benchmark

In this section is showed the performance and energy efficiency of the Jetson TK1 evaluation Board. For both benchmarks the NVIDIA CUDA nBody sample is used because it is more compute than memory intensive.

A. Performance

The performance of the Jetson TK1 is benchmarked against a system with an Intel Ivy Bridge E5-2630 v2 with two times 6 cores at 2.6 GHz, 64GB of RAM and a Tesla K20m. In

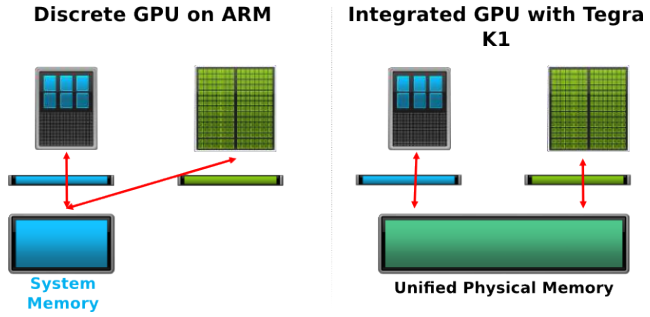

Fig. 10. Zero Copy with cache benefits [3]

the figure 11 can be seen that at a number of 1024 bodies the Jetson TK1 reaches his peak-performance. This is really good for small problem sizes. The Tesla K20m only reaches the peak performance at 16384 bodies.

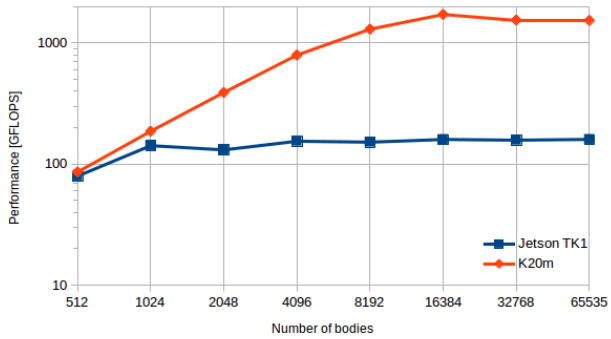

Fig. 11. Performance analysis between Jetson TK1 and Tesla K20m

The early reach of peak performance can be explained with figure 8. Due to a significant lower number of cores, the Jetson TK1 can work with full power already with 1024 bodies. The Tesla K20m is a Tesla GPU with 2496 CUDA cores instead of 192 CUDA cores like the Tegra K1 has. So to work with peak-performance the Tesla K20m needs 16384 bodies.

With a peek performance of 160 GFlops on Jetson TK1 versus 1722 GFlops on Tesla K20m Jetson TK1 reaches an eleventh part of performance with only a thirteenth part of cores, which is pretty impressive for this small device. Also that in the NVIDIA CUDA nBody sample the zero copy with cache benefits advantage is not used.

B. Efficiency

In the efficiency benchmark the Jetson shows its overall energy efficiency. The full system, consisting of power supply and the whole evaluation board. In bootup mode the system did use up to 6.5 Watts. When booting was finished and the system was idle it consumed only 3.2 Watts.

In default the Jetson TK1 is in energy saving mode. The GPU and CPU clock rates are on minimum. In the table I it

can be seen that the performance in the benchmark did only reach 13.4 single precision GFlops and an efficiency of 3.2 single precision GFLOPS per Watt was achieved.
Compared to nBody benchmark where the GPU clock rate is set to 852 MHz the achieved GFlops are much higher than in energy saving mode. It is more than ten times the value than before.

TABLE I. ENERGY CONSUMPTION OF JETSON TK1

System Status	Power [W]	GFlops	GFlops/W
boot	up to 6.5	-	-
idle	3.2	-	-
nBody (energy saving)	4.2	13.4	3.2
nBody (GPU max clock rate)	14.2	159.9	11.3
nBody on K20m (only GPU)	162	1753	10.8

Impressive is the achieved efficiency of 11.3 single precision GFlops per Watt compared to the 10.8 that the K20m achieved. On the K20m is only measured the GPU consumption. In figure 12 can be seen that NVIDIA is going to double the efficiency with Maxwell and quadruple it with Volta.

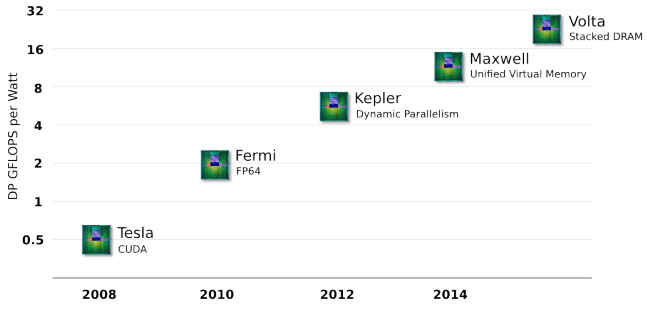

Fig. 12. NVIDIA GPU Roadmap [2]

This gain of efficiency is important for the 20 MWatt exaflop goal in 2020.[9]

V. RELATED WORK

The NVIDIA Tegra K1 is build for embedded systems, so many of other related work is using the Jetson TK1 as an embedded system and not as a high performance computing system. One interesting work is to use Jetson TK1 and Microsoft Kinetic to control a robotic system in a room. [10]

Another interesting Work is the AdasWorks Automated Driving [11]. The Jetson TK1 is controlling the whole car and gets the course information through two front facing cameras. It is processing the camera images and calculates the steering signals.

VI. CONCLUSION

In the paper it can be seen that the NVIDIA Jetson TK1 is a great step towards energy efficiency. The paper did show, that the NVIDIA Tegra K1 could be used as a high performance system. Further steps would be to build a cluster of Jetson TK1s and test their ability in using message pass functions. The related work shows that embedded GPGPU computing is a technology that will revolutionize the automation industry. Through the new gained performance in embedded image processing, technology is a step further to automation that can work like human eyes do and make it easier to use augmented reality or robotic systems.

VII. FUTURE WORK

The next steps to see if the Jetson TK1 is a great high performance component will be benchmarks with MPI and CUDA which also use the full capability of the shared physical memory. Furthermore a new board with support of the PCIe 2.0 4x for high performance interconnection networks to make use of MPI with low latency data movements.

REFERENCES

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[2] J. Purches and T. Lanfear, "NVIDIA technology overview," 2013. [Online]. Available: https://intranet.birmingham.ac.uk/it/teams/infrastructure/fm/bear/documents/public/CUDA-2013-07-31/NVIDIA-Technology-Overview.pdf

[3] A. Rao and N. Garg, "Mobile GPU compute with tegra k1," 2014. [Online]. Available: http://on-demand.gputechconf.com/gtc/2014/presentations/S4906-mobile-compute-tegra-K1.pdf

[4] L. Oden and H. Froning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2013, pp. 1–8.

[5] T. Lanfear, "CUDA tutorial," 2009. [Online]. Available: http://gpulab.compute.dtu.dk/PhDschool/slides/CUDA%20Tutorial.pdf

[6] NVidiaCorp, "CUDA c programming guide," 2014. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[7] NVidiaCorp., "NVIDIA jetson TK1 development kit," 2014. [Online]. Available: http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_platform_brief_May2014.pdf

[8] S. Yi, I. Yoon, C. Oh, and Y. Yi, "Real-time integrated face detection and recognition on embedded GPGPUs," in *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, Oct. 2014, pp. 98–107.

[9] B. Subramaniam, W. Saunders, T. Scogland, and W.-c. Feng, "Trends in energy-efficient computing: A perspective from the green500," in *Green Computing Conference (IGCC), 2013 International*, Jun. 2013, pp. 1–8.

[10] M. N. Rud and A. R. Pantiykchin, "Development of GPU-accelerated localization system for autonomous mobile robot," in *2014 International Conference on Mechanical Engineering, Automation and Control Systems (MEACS)*, Oct. 2014, pp. 1–4.

[11] "AdasWorks automated driving," Jan. 2015. [Online]. Available: https://www.youtube.com/watch?v=37cOQS9gc1w