VLSC: Language Proposal

David Chen, Kellie Ren Lu, Chance Steinberg, Diana Valverde
{dhc2129, krl2130, cws2136, drv2110}@columbia.edu

Language guru: Diana Valverde
System architect: Kellie Ren Lu
Verification and Validation: David Chen
Manager: Chance Steinberg

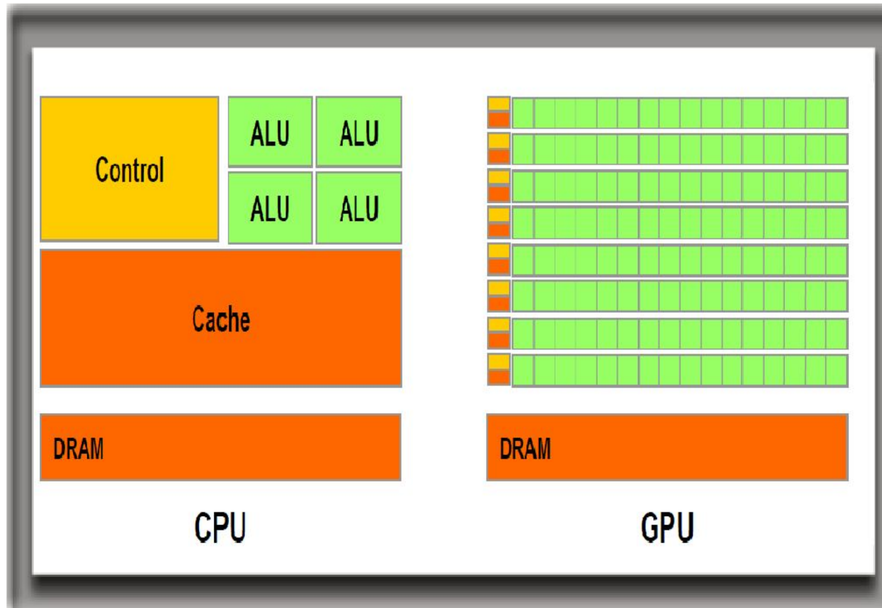
VLSC is a high level language, similar to Python, intended to operate on Graphical Processing Units (GPUs). The primary implementation for VLSC is for numerical computation, which can be performed orders of magnitude faster on GPU architecture than on x86. Other functionality provided by VLSC includes map/reduce, and filtering. VLSC is intended to provide convenient and safer access to the GPU's computational power by circumventing some of the lower level minutiae of CUDA.

VLSC closest analog in the universe of high level GPU front end languages is pyCuda: (https://documen.tician.de/pycuda/index.html#). A language that acts as a functional subset of pyCuda that compiles to LLVM or PTX is the goal of VLSC.

Graphics processing units (GPUs) are specialized processors that leverage parallel processing to complete tasks at high speeds. The performance boost is accomplished by dedicating most of the circuitry to multi-threaded arithmetic logic units (ALUs or cores). This is done at the expense of program control and internal memory, making the GPU less useful as computation becomes more complex.

"The GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - -the ratio of arithmetic

operations to memory operations”
(http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#ixzz3zYLypdbk)



Source: http://www.keremcaliskan.com/wp-content/uploads/2011/01/CPU-GPU-Structures1.png

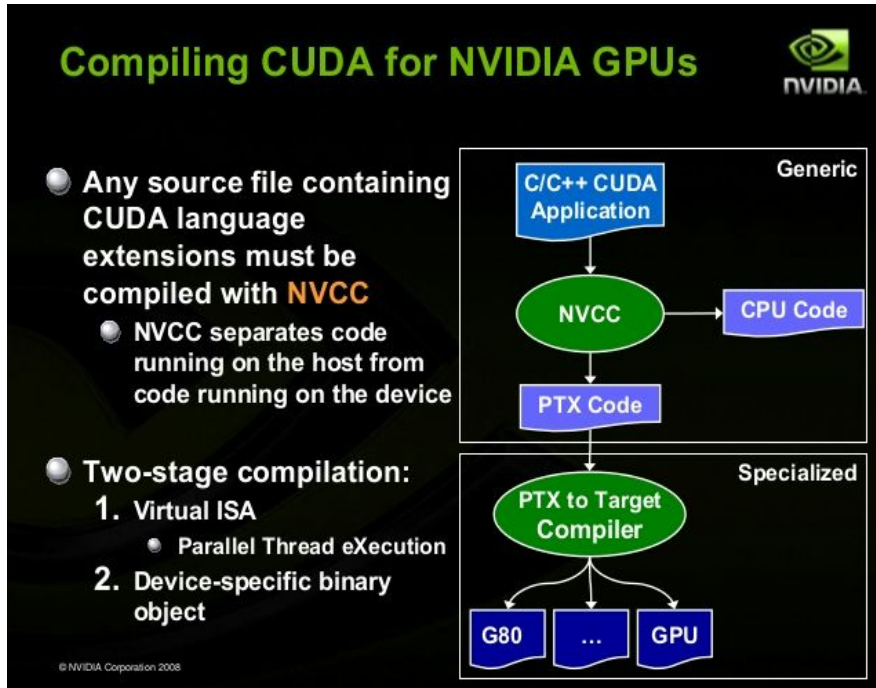
NVIDIA developed the CUDA framework to work with their GPU processors, and there are several languages that interface with the CUDA framework, for example CUDA C (http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3zYKu0Mny)

VLSC provides some of the functionality of a CUDA compatible language like CUDA C, but unlike CUDA C, it is designed to carry out certain GPU specific tasks with relatively few commands.

Low level language options

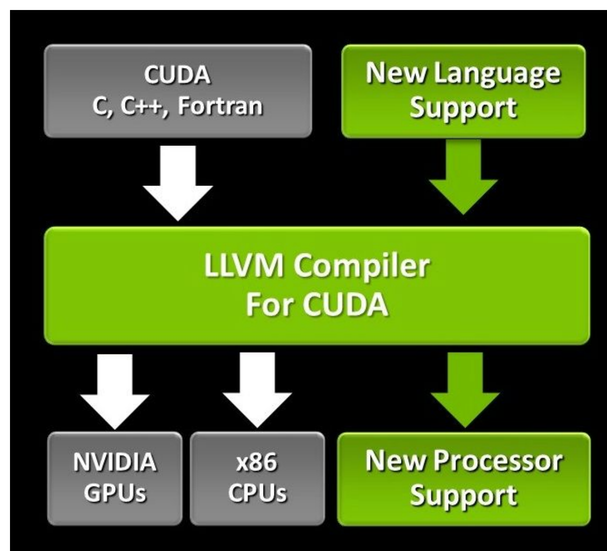
Compiling VLSC to a lower level language allows for specialized functionality and optimization choices that are not currently available in existing CUDA libraries.

PTX (parallel thread execution) is an assembly like language that operates on GPUs. PTX operates on threads within a parallel thread array: the thread array is at the heart of the GPUs parallel processing power. CUDA coded applications are compiled to PTX using the NVCC framework as shown below:



Source:
http://image.slidesharecdn.com/nvidiacudatutorialnondaapr08-131016063306-phpapp01/95/nvidia-cuda-tutorialnondaapr08-31-638.jpg?cb=1381905231

VLSC compiles directly to PTX, replacing the CUDA / NVCC processing entirely. LLVM is an alternative to PTX that is lower level than CUDA, but easier to implement. LLVM based compilers already exist for CUDA applications, which could provide for an easier testing environment.



http://image.slidesharecdn.com/nvidiacudatutorialnondaapr08-131016063306-phpapp01/95/nvidia-cuda-tutorialnondaapr08-31-638.jpg?cb=1381905231

Syntax

//This is a comment

/*
This is a multi-line comment
*/

Primitive declarations

int	Default: 32 bit signed integer Other: (maybe?) unsigned 32-bit
float	Single-precision 32-bit IEEE 754 floating point
double	Double-precision 64-bit IEEE 754 floating point
boolean	true or false

Arrays

- Objects that hold a fixed number of values of a single type

int[] myArray	1-Dimensional array of integers
int[][] myArray	2-Dimensional array of integers
int[][]...[] myArray	n-Dimensional array
int[10] myArray = {0}	Initialize myArray with 10 0's.
int[10] myArray = {*}	Initialize myArray with 10 random integers
int[10] myArray = {0...9}	Initialize myArray with 10 sequential integers
block(myArray, n)	Creates a 2d array out of myArray by blocking n-elements

n-dimensional arrays are distinct from arrays of arrays for memory mapping/access. Arrays of arrays may or may not be implemented depending on whether we think it's important to include arrays of variable length arrays in our grammar.

Arithmetic and Logical Operators

+, -, /, *, %	- Vector parallel addition, subtraction, division, multiplication and modulo - Arithmetic operations on primitives.
.	Dot product, accepts only 1-dimensional arrays
**	Matrix multiplication, accepts only 2-dimensional arrays
log(myArray, n) log(myArray, n, floor)	Takes the log or floored log of every element in base n, useful when dealing with very large numbers.
myArray^n	Raises every element in array to the nth power
&&, ||, !=, !, ==, XOR	Standard logic operators
<<, >>	Bit-shift operators
concatenate(a1, a2)	Concatenates a1 and a2, accepts only 1-dimensional arrays
copy(element, n)	Copy element n-times

Main function

-Python syntax: def main():

Functions and Closures:

- Python-like syntax
- If called within map, reduce or filter, the function can access all variables defined in the scope in which it's called.

```
def <return type> <function name>(arg1, arg2…):
        return f(arg1, arg2, …)
```

e.g.

```
def int add(int x, int y):
        return x + y
```

Built-in Higher-Order Functions

Two first-class functions:

- map(f, arg1, …, argn)
    - map(f, v1, v2) = v3, where v3[i] =f(v1[i], v2[i])
    - map(f, m1, m2) = m3, where m3[i][j] = f(m1[i][j], m2[i][j])

    Example: map(add, v1, v2)
- reduce(f, arg)
    - Applies function to argument using a parallel tree-based approach.
    - reduce(add, v1) returns the sum of all elements in v1.
- filter(f, arg)
    - Returns all elements from a list that evaluate True when passed to a certain function

    Example:

        def boolean divisible(int x): return x%2 == 0

        filter(divisible, v1)

Using map, reduce and filter a user can specify any parallel operation with closures.

Control Flow

- Will use python syntax for for loops, if/else if/else statements and while loops.

Special data types

- To simplify CPU instructions, strings will only be used as arguments to predefined I/O stream functions.

        Example: print("Hello World")
                 error("Argument expected")

Example program

```
//Parallel encryption using CTR block cipher
#import sys

def boolean[] bit_representation(int a):
        boolean[32] binary
        for(int i = 0; i < 32; i++):
                binary[i] = (a & (1 <<i)) != 0

        return binary

def main():
        int[] KEY = sys.in(arg[1]) //takes in integers
        int[] plaintext = sys.in(arg[2]))

        int[128] nonce = {*}
        int[10] counter = {1...10}

        int[][] nonce_ = copy(nonce, 10)
        int[][] counter_ = map(bit_representation, counter)
        int[][] combined = map(concatenate, nonce_, counter_)

        //AES combined with KEY, assume AES function exists
        int[][] cipher = map(AES, combined, KEY)

        int[][] plaintext_ = block(plaintext, 10)
        ciphertext[][] = map(XOR, cipher, plaintext_)

        output  = reduce(concatenate, ciphertext)

        print(output)
```