Machine Programmer Documentation

November 9, 2016

Contents

Chapter 1

Readme

1.1 Abstract

Make specifications for your programs, and allow the computer to evolve the solutions.

1.2 Ingredients

This project has several components:

- binary/encoding for encoding Pyash into 32 byte tablets. (beta)
- binary/clprobe for getting OpenCL info and compiling .cl files to check for syntax errors (beta)
- Machine programmer for evolving Pyash programs in OpenCL on GPU/CPU (alpha)
- OpenCL compatible virtual machine for Pyash, the SPEL core-language (alpha)
- Compiler for converting Pyash byte-code to other languages like LLVM (concept)

1.3 Progress

As of Aug 2016, this is just a prototype under active development. It is expected that once the Machine Programmer can contribute to it's own code base, that the rate of development will increase.

1.4 Tiny OpenCL Teaching

Even now it has some useful AGPLv3, OpenCL code, which can be adapted to other projects. check out the following files for a mini OpenCL overview:

```
source/hello.c
source/hello.cl
source/generic.h
source/generic.c
```

can test with

```
cd source
gcc generic.c hello.c -lOpenCL -o hello # possibly also
   -L../library
./hello
```

Chapter 2

Installation

2.1 Introduction

This is a guide for how to install Machine Programmer on Ubuntu Linux[**Ubuntu**]. Ubuntu Linux is perhaps the most popular Linux Distribution, also it is free to download and install (though donations are welcome).

If you have a different operating system such as those made by Microsoft or Apple, feel free to get it working in whatever method you use, and send corresponding documentation so it can be added as an option.

The version of Ubuntu Linux this tutorial will be refering to is 16.04, though at least for proprietary AMD drivers, you may have better luck on 14.04 or Fedora.

If you use a different version of Linux, I'm sure can adapt the package names to those which are suitable for your distribution. If you'd like to document how you did it then can forward it and will add your manual also.

2.2 Dependencies

As with any package the most difficult part is getting the dependencies installed.

needs autotools to compile

```
apt-get install autoconf automake libtool build-essential
```

2.2.1 Documentation

If you desire to compile the documentation.

```
apt-get install texlive-latex-base texlive-latex-recommended
   texlive-science \
biber
```

2.2.2 OpenCL

The hardest part of compiling this is probably installing OpenCL. If you have that down, can simply:

```
./autogen.sh && ./configure && make && binary/programmer
```

Otherwise if that doesn't work, then you probably need OpenCL, so follow along below.

2.2.3 CPU OpenCL Installationdownload opencl header packages

If you'd like to take advantage of your CPU cores, or you don't have an OpenCL compatible GPU, then you'll have to install Portable Open Compute Library or POCL.

POCL has some dependencies on Ubuntu

```
apt-get install libhwloc-dev zlib1g-dev libclang-dev libx11-dev
   ocl-icd-dev
cmake opencl-headers llvm clang
```

Then if you are sure you don't have GPU drivers, as installing this package may conflict with any GPU drivers you may install:

```
apt-get install ocl-icd-opencl-dev
```

then compile POCL based on it's instructions, if it's being difficult, can try the cmake part of the instructions, as cmake may give you additional dependencies.

Also note that you need to have likely the most recent OpenCL headers available, as of this writing, pocl-0.13 needs opencl 2.0 headers.

If you find that the dependencies I listed about are insufficient, then please email me with how you got it working. Otherwise once you have POCL can simply

```
./autogen.sh && ./configure && make && binary/programmer
```

2.2.4 GPU OpenCL Installation

For all GPU's the supported version of OpenCL may differ from the latest version. To avoid deprecation warnings and unexplained segmentation faults, can install the header files which are pertinent to your supported OpenCL version.

You can find out what version of OpenCL your hardware supports either from it's documentation, or by successfully installing OpenCL and then running the hello scripts, which will also give you information about your available OpenCL platforms.

For your convenience I've included zip files of the headers in library/. For example if your GPU supports OpenCL 1.1 then can install by doing:

```
cd library/
unzip OpenCL-Headers-opencl11.zip
mv OpenCL-Headers-opencl11 CL
sudo mv CL /usr/local/include/
```

(ARM) Mali GPU's (O-Droid XU3/4)

So far have tested it with the Mali OpenCL SDK, which works on the ODroid, an open-hardware heterogenous processing SoC board.

To run it need to do

```
apt-get install mali-x11 # and probably restart X server.
```

Because the standard opencl-headers provided by Ubuntu are 2.0+, and the Mali-T628 only supports up to 1.1 you may have to copy the CL folder from Mali SDK to /usr/local/include, to avoid a bunch of deprecation warnings and-or errors.

```
sudo cp -rv include/CL /usr/local/include
```

Strangely enough, in order to get it to compile, and to get it to run requires different versions of libOpenCL.so. However it seeems if POCL is installed, then can simply compile now with:

```
./autogen.sh && ./configure make && binary/programmer
```

Otherwise if POCL is Not installed the commands would be:

Mali's libOpenCL.so installation

In order to compile need the ones from Mali OpenCL SDK

After downloading that package, extract it and fix up platform.mk to reflect your current platform. For example:

```
CC:=arm-linux-gnueabihf-g++
AR=arm-linux-gnueabihf-ar
```

then in the Mali SDK folder compile the libOpenCL.so

```
cd lib/ && make
```

once you have a libOpenCL.so can put it into the machine-programmer's library/ folder.

```
cp lib/libOpenCL.so $MACHINE_PROGRAMMER_PATH/library/
```

```
./autogen.sh && ./configure LDFLAGS=-L./library && make &&
   binary/programmer
```

Intel GPUs

Intel includes onboard GPUs on a lot of motherboards, so chances are good that if you have an Intel CPU, that you may also have an intel GPU, which you can take advantage of using the well functioning open source beignet drivers.

```
apt-get install beignet beignet-dev beignet-opencl-icd
```

Note this does not work with the AMD hardware that I've tested it with.

Nvidia GPUs

Unfortunately there are no libre drivers for Nvidia that have OpenCL support, however there are proprietary drivers, which may work in certain cases. Note this means you can't use UEFI, common on modern laptops, you'll have to disable it in the bios. Also this could make breaking changes so I advise you backup your data before attempting to install proprietary drivers.

Before you begin, make sure your system is up to date;

```
sudo apt-get update && sudo apt-get upgrade && sudo apt-get
    dist-upgrade;
sudo apt-get autoremove #cleans up extra packages
```

There are a variety of versions of the nvidia drivers, this is because they are very finicky and they might not all work with your GPU card. For instance I had to try several, and spend several days testing, before I finally figured out which ones worked.

The best one for my GeForce GTX 960M, is unknown-361. I tried their recommended nvidia-367 and nvidia-370 those didn't work, and nvidia-340 didn't even show any picture on the screen (good thing I had ssh).

So I would advise you backup your system, and install ssh, so you could ssh tunnel into your computer and fix it in case the monitor stops working due to incompatible drivers.

Another good troubleshooting method when the screen is blank due to proprietary drivers, is to hold the shift-key during boot-up as this can let you enter the grub menu, where you can select "Advanced options" and use a recovery mode or different kernel.

Also may be a good idea to have a boot recovery disk or installation disk handy, because installing proprietary drivers may make the system unbootable.

At first reboot after installing the drivers I noticed that the computer turned itself off and on several times before loading properly, I'm guessing this is normal for prorpeitary drivers.

You may find it doesn't work well with the standard dev packs, something I was struggling with for days. But fortunately can install the open source ones from Intel.

```
apt-get install beignet-opencl-icd
```

You may also run into it not using the proprietary drivers, for instance if you do

```
lsmod |grep nvidia
```

it may give you nothing.

in this case may need to update-alternatives

```
sudo update-alternatives --config x86_64-linux-gnu_gl_conf
```

This however wont work if you don't have any working proprietary drivers.

I manged to get nvidia proprietary-drivers working for one or two boots, however I was not able to reliably reproduce it.

Other GPU's

You would have to see what is available for your platform, if you have success in getting OpenCL, then please write up a summary and email me.

2.3 Developing

If you would like to help with development, need some additional packages.

```
apt-get install clang-format git
```

2.4 Contact

Can email me at streondj@gmail.com for details.

Chapter 3

binary/clprobe

OpenCL Probe Logan Streondj November 9, 2016
Version 0.1

Abstract

Demonstrates OpenCL platform device knowledge xor tries to compile a .cl file based on input filename.

3.1 Synopsis

clprobe input filename

3.2 Description

if there is no filename ingredient then *clprobe* demonstrates openCL which includes the platform and instrument knowledge.

```
  if (ingredient_long != 2) {
    fprintf(stderr,
            "no .cl filename ingredient so demonstrating openCL knowledge\n");
    getInfo();
    exit(0);
  }
```

Otherwise Progclprobe accepts one ingredient, which is the input filename.

```
  char *filename = ingredient_list[1];
```

It then sets up a minimal OpenCL environment, and attempts to compile it.

```
  cl_platform_id platform_id = NULL;
  cl_uint ret_num_platforms;
  cl_context context = 0;
  cl_program program = 0;
  cl_device_id device_id = 0;
  cl_uint ret_num_devices;
  cl_int return_number;

  return_number = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
  if (!success_verification(return_number)) {
    fprintf(stderr, "Failed to get platform id's. %s:%d\n", __FILE__, __LINE__);
    return 1;
  }
  return_number = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
                                 &device_id, &ret_num_devices);
  if (!success_verification(return_number)) {
    fprintf(stderr, "Failed to get OpenCL devices. %s:%d\n", __FILE__,
            __LINE__);
    return 1;
  }
  context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &return_number);
  if (!success_verification(return_number)) {
    fprintf(stderr, "Failed to create an OpenCL context. %s:%d\n", __FILE__,
            __LINE__);
    return 1;
  }
  seed_program_probe(device_id, context, filename, &program);
```

3.3 Bugs

Still need to add ability to load header files.

3.4 Author

Logan Streondj ¡streondj at gmail dot com¿

3.5 See Also

generic.c *libOpenCL*(7)

3.6 License

Chapter 4

Pyash Encoding

The virtual machine uses variable-length-instruction-word (VLIW), loosely inspired by head and tails instruction format (HTF). HTF uses VLIW's which are 128 or 256 bits long, however there can be multiple instructions per major instruction word.

4.1 VLIW's Head Index

The head is really a parse index, to show the phrase boundaries. In TroshLyash each bit represents a word, each of which is 16bits, when a phrase boundary is met then the bits flip from 1 to 0 or vice-versa at the phrase boundary word. index takes up the first 16bits of the VLIW. This would lead to 256bit (32Byte) VLIW's. The real advantage of the indexing occurs when there either multiple sentences per VLIW, or when there are complex sentences in the VLIW's. Having the VLIW's broken up into 32Byte chunks, makes it easier to address function entry points, which can be placed at the beginning of a VLIW. Can fit 16 VLIWS in a POSIX page, 128 VLIW's in a Linux page, so would only need 1 byte (8bits) for addressing functions that are within 1 page distance.

4.2 Word Compression

Now for the slightly more interesting issue of packing as many as 5 glyphs into a mere 16 bits. Why this is particularly interesting is that there is an alphabet of 32 glyphs, which would typically required 5 bits each, and thus

25bits in total. However the 16 bit compression is mostly possible due to the rather strict phonotactics of TroshLyash, as only certain classes of letters can occur in any exact place. The encoding supports 4 kinds of words, 2 grammar word classes and 2 root word classes. Where C is a consonant, T is a tone and V is a vowel, they are CVT, CCVT, and CVTC, CCVTC respectively.

4.2.1 CCVTC or CSVTF

I'll start with explaining the simplest case of the CCVTC word pattern. To make it easier to understand the word classes can call is the CSVTF pattern, where S stands for Second consonant, and F stands for Final Consonant. The first C represents 22 consonants, so there needs to be at least 5 bits to represent them. Here are the various classes

"C"	:"p","t","k","f", "s","c","x", "b","d","g","v", "z","j", "n","m","q","r", "l","y","w",
"S"	"f","s","c","y", "r","w","l","x", "z","j","v",
"V"	"i","a","e","o","u","6",
"T"	"7","_",
"F"	"p","t","k","f", "s","c","n","m"

, (can check the phonology page for pronunciation) C needs 5 bits, S would need 4 bits, however the phonotactics means that if the initial C is voiced, then the S must be voiced, thus "c" would turn into "j", "s" into "z" and "f" into "v", also none of the ambigiously voiced phonemes (l, m, n, q, y, w, r) can come before a fricative because they have a higher sonority, thus must be closer to the vowel. So S only needs 3 bits. V needs 3 bits T needs 2 bits and F needs 3 bits which is a total of 16 bits. 5+3+3+2+3 = 16 However there are other kinds of words also. we'll see how those work.

4.2.2 HCVTF

So here we have to realize that CVC or CVTC is actually HCVTF due to alignment. So what we do is make a three bit trigger from the first word, the trigger is 0, which can be three binary 0's, 0b000 3+5+3+2+3 = 16

H+C+V+T+C this does mean that now 0b1000, 0b10000 and 0b11000 is no longer useable consonant representation, however since there are only 22 consonants, and only 2 of those are purely for syntax so aren't necessary, so that's okay, simply can skip the assignment of 8, 16 and 24.

4.2.3 CSVT

This is similar to the above, except we use 0b111 as the trigger, meaning have to also skip assignment of 15, 23 and 31. $3+5+3+3+2 = 16$?+C+S+V+T

4.2.4 CVT

For this one can actually simply have a special number, such as 30, which indicates that the word represents a 2 letter word. $5 + 5 + 3 + 2 + 1$ $F + C + V + T + P$ what is PF P can be a parity-bit for the phrase, or simply unassigned.

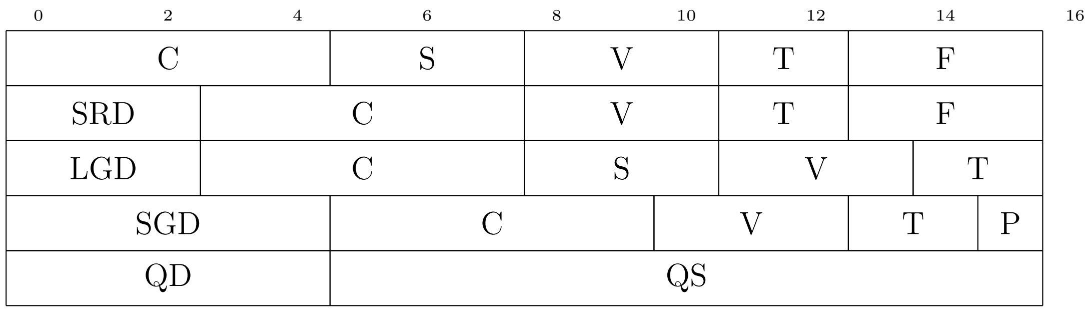
4.3 Quotes

Now with VM encodings, it is also necessary to make reference to binary numbers and things like that. The nice thing with this encoding is that we can represent several different things. Currently with the above words, we have 1 number undefined in the initial 5 bits. 29 can be an initial dot or the final one, can call the the quote-denote (QD), depending on if parser works forwards or backwards. Though for consistency it is best that it is kept as a suffix (final one), as most other things are suffixes. 5+3+8 = 16 Q+L+B QD has a 3 bit argument of Length. The Length is the number of 16bit fields which are quoted, if the length is 0, then the B is used as a raw byte of binary. Otherwise the B represents the encoding of the quoted bytes, mostly so that it is easier to display when debugging. The type information is external to the quotes themselves, being expressed via the available TroshLyash words. So in theory it would be possible to have a number that is encoded in UTF-8, or a string that is encoded as a floating-point-number. Though if the VM interpreter is smart then it will make sure the encoding is compatible with the type Lyash type, and throw an error otherwise.

4.4 Extension

This encoding already can represent over 17,000 words, which if they were all assigned would take 15bits, so it is a fairly efficient encoding. However the amount of words can be extended by increasing number of vowels, as well as tones. And it may even be possible to add an initial consonant if only one or two of the quote types is necessary. However this extension isn't likely to be necessary anytime in the near future, because adult vocabulary goes up to around 17,000 words, which includes a large number of synonyms. For instance the Lyash core words were generated by combining several different word-lists, which were all meant to be orthogonal, yet it turns out about half were internationally synonyms, so were cut down from around eight thousand to around four thousand words. It will be possible to flesh out the vocabulary with compound words and more technical words later on. Also it might make sense to supplant or remove some words like proper-names of countries.

4.5 Encoding Tidbit Overview



Legend C	Initial Consonant
S	Secondary Consonant
V	Vowel
T	Tone
F	Final Consonant
SRD	Short Root Denote
LGD	Long Grammar Denote
SGD	Short Grammar Denote
P	(optional) Phrase Parity Check tidbit
QD	Quote Denote
QS	Quote Sort **??**

4.6 Table of Values

#	C	S	V	T	F
width	5	3	3	2	3
0	SRD	y	i	MT	m
1	m	w	a	7	k
2	k	s z	u	_	p
3	y	l	e	U	n
4	p	f v	o		s
5	w	c j	6		t
6	n	r	U		f
7	LGD	x	U		c
8	SRO				
9	s				
10	t				
11	l				
12	f				
13	c				
14	r				
15	LGO				
16	SRO				
17	b				
18	g				
19	d				
20	z				
21	j				
22	v				
23	LGO				
24	SRO				
25	q				
26	x				
27	1				
28	8				
29	QD				
30	SGD				
31	LGO				

	blank means out of bounds
U	undefined
MT	middle tone, no marking
QD	quote denote
SGD	short grammar word denote
SRD	short root word denote
LGD	long grammar word denote
SRO	short root word denote overflow
LGO	long grammar word denote overflow

4.7 Quote Sort

0	5	7	10	12 15
	QS			
5	3	2	2	4
QD	VT	R	TW	SD

4.7.1 definitions

QS quote sort

QD quote denote

VT vector thick

R region

ST scalar thick

SD sort denote

16 tidbit									
0	4	5	6	7	9	10	11	12	15
5 tidbit		2 tidbit		3 tidbit		2 tidbit		4 tidbit	
quote denote		ingredient thick		vector thick		region		sort denote	

definitions				
0	1 byte, 8 tidbit	1	literal	glyph
1	2 byte, 16 tidbit	2	global	word
2	4 byte, 32 tidbit	4	constant	phrase
3	8 byte, 64 tidbit	8	local	sentence
4		16		text
5		U		function
6		U		database
7		3		named database
8				unsigned integer
9				signed integer
A				floating point number
B				U
C				U
D				U
E				U
F				U

The quote denote is 5 bits long, leaving 11 bits. the next 2 bits is used to indicate bit width of quote ingredient (s), the following 3 bits is used to indicate the number of vector ingredient (s), 2 bit for region

1. literal
2. global memory referential
3. constant memory referential
4. local memory referential

4 bits for noun denote:

1. letter
2. word

3. phrase
4. sentence
5. text
6. function
7. datastructure
8. named data type, next 16 bits gives name
9. unsigned integer
10. signed integer
11. floating point number

4.8 Denote Syntax

For literals

letter l _letter

word word _word

phrase word _acc _phrase

sentence word _acc _rea _independent_clause

text

function

datastructure

named data type

unsigned integer one two three _number (291)

signed integer one two three _negatory_quantifier _num (-291)

floating_point_number two four _floating_point_num ten _bas one _neg _exponential _num (2.4)

	0 source-case	1 location-case	2 destination-case	3 way-case
0 base	nominative-case	accusative-case	dative-case	instrumental-cas
1 space-context (x)	ablative-case	locative-case	allative-case	prosecutive-case
2 genitive-case	possessive-case	relational-case	possessed-case	descriptive-case
3 discourse-context	initiative-case	vocative-case	terminative-case	topic-case
4 social-context	causal-case	comitative-case	benefactive-case	evidential-case
5 surface-context (y)	delative-case	superessive-case	superlative-case	vialis-case
6 interior-context (z)	elative-case	inessive-case	illative-case	perlative-case
7 time-context (t)	initial-time	temporal-case	final-time	during-time

Table 4.1: grammtical-case number system

fixed_point_number two _flo one _num (2.1)

rational one _rational three _num (1/3)

decimal number ten _bas one one _num (11)

hexadecimal number sixteen _bas eleven _num (11)

vector world _word _and _voc _word two sixteen word _vector (vector of 16 unsigned shorts each short containing a word, intialized to repeating sequence of "hello _vocative_case")

4.9 Independent-Clause Code Name

Each independent-clause can have a code name to help find it's program.

There is a mix of grammatical-cases, sorts and a verb in each independent clause. For matching with modern computer processing, a 64bit thickness is desired, though a 128 bit thickness may be possible.

The grammatical cases can have a table to make it easy to identify them.

five bits to designate the case, and 11 bits for the quote type.

The context will henceforward be referred to as scene, and the other half being the posture.

This does mean that any independentClause would have a maximum of three grammatical-cases plus a verb if in 64 bit.

Can also have seperate identifiers for the verb and the grammatical-cases, then it would be easier to have multi-word verbs.

0 2 3 4 5 6 7 9 10 11 12 15

2	3	2	3	2	4
P	S	QS			

Table 4.2: grammtical-case code

P posture

S scene

QS quote sort **??**

0 2 4 6

64 tidbit			
G	G	G	V

0 2 4 6

VUL2 vector of two 64 tidbit numbers for searching			
or VUS8 vector of eight 16 tidbit numbers for establishing			
G	G	G	G
V	V	V or A	Pe

Table 4.3: code name sketch

G grammatical-case

V verb

A aspect

Pe perspective (grammatical-mood)

Chapter 5

binary/encoding