Relatório de aceleração de uma aplicação com o Framework LiteX

Bruno Sobreira França (217787)
Luis Felipe Lapa Barcelos Coutinho (182956)

1 de julho de 2025

Conteúdo

1 Introdução

Esse relatório descreve os passos tomados para desenvolver um acelerador utilizando o framework LiteX e programar uma placa Tang Nano 9K para executar o código acelerador. São descritos o processo de configuração do ambiente, a escolha da aplicação a ser acelerada, o método utilizado para medir o tempo de execução da aplicação, o desenvolvimento de um acelerador para a aplicação e os resultados obtidos comparando o tempo de execução com e sem o uso do acelerador.

2 Configurando o ambiente LiteX

O primeiro passo consistiu em preparar o ambiente de desenvolvimento, instalando o framework LiteX e suas dependências e definindo as variáveis de ambientes a serem utilizadas.

2.1 Instalando LiteX

A seguir estão os comandos utilizados para a instalação do LiteX em um ambiente Linux. Para mais informações sobre processo de instalação acessar [1].

1. **Download do script de configuração do LiteX:**

```
$ wget https://raw.githubusercontent.com/enjoy-digital/litex/
   master/litex_setup.py
```

2. **O script foi tornado executável:**

```
$ chmod +x litex_setup.py
```

3. **Inicialização e instalação do LiteX e suas dependências:**

```
$ ./litex_setup.py --init --install
```

2.2 Intalaçao do toolchain para RISC-V

Caso o SoC desenvolvido com o LiteX inclua uma CPU RISC-V (por exemplo o VexRiscv) é necessário instalar uma toolchain capaz de compilar o software que será executado na CPU RISC-V.

```
$ pip3 install meson
$ ./litex_setup.py --gcc=riscv
```

2.3 Definindo o caminho raiz do LiteX

Para garantir que o LiteX pudesse localizar seus vários componentes e para facilitar a navegação nos projetos, foram definidas uma variáveis de ambiente apontando para o diretório raiz do LiteX e para uma pasta de build. (O caminho `"/home/user/mo801/"` é um exemplo.)

```
$ export LITEX_PATH=/home/user/mo801/litex
$ export BUILD_DIR_ROOT=/home/user/mo801/litex_build
```

2.4 Programando a placa Tang Nano 9K

Esta subseção detalha como o LiteX foi utilizado para montar e programar um SoC na placa Tang Nano 9K.

2.4.1 Instalação do toolchain da placa

Antes de programar a FPGA, foi preciso instalar as toolchains e utilitários de programação apropriados. Para a Tang Nano 9K, que utiliza uma FPGA Gowin, as ferramentas necessárias são:

- **OSS-CAD Suite:** Um conjunto de ferramentas de código aberto para desenvolvimento de FPGA. Inclui ferramentas essenciais como `yosys` para síntese, `nextpnr` para Place and Route, `openFPGALoader` e `Apicula` para programar FPGAs.
- **Gowin Educational Toolchain:** Embora o OSS-CAD Suite forneça alternativas de código aberto, a Gowin também oferece sua toolchain oficial. Essas ferramentas podem ser necessárias para alcançar um desempenho ideal.

O processo de instalação dessas ferramentas mostrou-se complicado. Para usuários de Linux, um obstáculo comum é configurar as permissões USB corretas para o FPGA loader. O script abaixo é útil para configurar as regras `udev` necessárias.

```
$ curl -sSL https://raw.githubusercontent.com/lushaylabs/
   openfpgaloader-ubuntufix/main/setup.sh | sh
```

O tutorial [2] contém instruções mais detalhadas sobre o processo de instalação das ferramentas OSS-CAD Suite e Gowin.

2.4.2 Build e programação do SoC na placa

1. **Build do SoC para a placa Tang Nano 9K:** Este comando sintetiza o SoC desenvolvido com o LiteX para a FPGA alvo.

```
$ ./sipeed_tang_nano_9k.py --build --output-dir=
   $BUILD_DIR_ROOT/sipeed_tang_nano_9k
```

2. **Programação da placa Tang Nano 9K com o bitstream gerado** Este comando utiliza a ferramenta configurada pelo LiteX para programar a FPGA com o bitstream gerado.

```
$ ./sipeed_tang_nano_9k.py --load --output-dir=
   $BUILD_DIR_ROOT/sipeed_tang_nano_9k
```

Caso a programação da placa tenha sido concluída com sucesso, os LEDs da placa vão exibir um comportamento de ring counter.

3 Aplicação acelerada: Regressão Logística Multinomial (RLM)

A aplicação escolhida para aceleração é a regressão logística multinomial (RLM) [3], uma extensão da regressão logística tradicional para problemas de classificação com múltiplas classes. Esse método estatístico é amplamente utilizado em tarefas onde o objetivo é prever a qual entre várias categorias uma determinada entrada pertence. Exemplos incluem

a classificação de imagens em diferentes objetos (como "gato", "cachorro", "carro", etc.) ou a categorização de documentos por tópico.

Na regressão logística binária, uma forma mais simples do modelo [4], o resultado é uma probabilidade entre duas classes. Já na versão multinomial, temos um conjunto de vetores de pesos $\mathbf{w}_j$ e vieses b_j, um para cada classe j. A predição é realizada por meio do cálculo do produto escalar entre a entrada $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ e cada vetor de pesos $\mathbf{w}_j = [w_j1, w_j2, \ldots, w_jn]$, adicionando o viés correspondente. A classe predita é aquela que resultar na maior pontuação, usando a função `argmax`:

$$\text{classe\_predita} = \arg\max_j \left(\mathbf{w}_j \cdot \mathbf{x} + b_j\right)$$

O processo de utilizar o modelo para a classificação final é conhecido como **inferência**. Ele ocorre após o **treinamento** do modelo, etapa em que os pesos $\mathbf{w}_j$ e os vieses $\mathbf{b}_j$ são ajustados com base em um conjunto de dados rotulado, utilizando algoritmos de otimização como o gradiente descendente. O treinamento é feito, normalmente, em servidores com alta capacidade computacional e não é o foco da aceleração.

Após a conclusão do treinamento, o modelo passa a ser utilizado para realizar previsões em tempo real. Essa fase de **inferência** é a mais executada na prática, especialmente em ambientes embarcadas (alvo desse trabalho), onde substituir o modelo em execução pode não ser trivial. Por esse motivo, é essa etapa que buscamos acelerar com hardware dedicado. A inferência é feita realizando o **produto interno (Dot product)** entre os vetores de entrada e o vetor de pesos do modelo. O produto interno entre dois vetores é computado realizando o produto termo a termo entre os elementos dos vetores seguido da soma dos produtos obtidos. A independência entre os produtos de termos diferentes abre a possibilidade para a computação diretamente no hardware de múltiplos produtos em paralelo.

4 MicroMLgen e Funções de inferência de RLMs em C

Como explicado anteriormente, o objetivo deste trabalho é acelerar uma regressão logística multinomial (RLM) para execução em um ambiente embarcado. No entanto, antes de realizar essa aceleração, é necessário treinar o modelo em um ambiente com maior capacidade computacional (no caso deste trabalho, um computador de pessoal) para, em seguida, portar seus parâmetros para um arquivo C que por sua vez vai ser compilado e executado na plataforma alvo, que nesse trabalho é a placa Tang Nano 9K.

Para isso, utilizou-se a biblioteca `MicroMLgen` [5], uma ferramenta que permite converter modelos treinados em Python, usando bibliotecas como `scikit-learn`, para implementações em linguagem C++. Essa conversão gera uma função de inferência que incorpora diretamente os pesos $\mathbf{w}_j$ e os vieses $\mathbf{b}_j$ aprendidos durante o treinamento, eliminando a necessidade de bibliotecas externas ou operações complexas em tempo de execução.

Um exemplo de código gerado pode ser visto na Figura 1, referente a um modelo de RLM treinado com o dataset de dígitos padrão do scikit-learn [6]. Esse dataset contém 10 classes a serem preditas, correspondentes aos algarismos arábicos de 0 a 9. Os pesos do modelo são passados como parâmetros para a função `dot`, responsável por computar o produto escalar entre esses pesos e os dados de entrada. Já os vieses são instanciados

diretamente no vetor votes, que contém 10 posições, uma para cada classe. Por fim, ao final da função, um laço é responsável por computar o argmax, ou seja, identificar a classe com a maior pontuação. Nota-se então, que a função gerada, mapeia diretamente o algoritmo de inferência do modelo, portanto ela é usada como baseline para realização de benchmarks.

Figura 1: Arquivo C++ gerado pela biblioteca Python `MicroMLgen`. A função `product` contém os pesos e vieses do modelo e chama a função `dot` para realizar o produto escalar. As funções `predictLabel` e `idxToLabel` convertem o índice numérico da classe predita para seu nome e vice-versa.

Para uma integração mais suave com o ambiente de desenvolvimento do LiteX, optou-se por extrair as funções geradas pelo MicroMLgen e incorporá-las diretamente em um arquivo **.c** para ser utilizado para realização de benchmarks.

5 Benchmark: Configuração do sistema e metodologia de medição

Para realizar as medições do tempo de execução da inferência ou de suas versões aceleradas, adotou-se uma abordagem baseada em contadores de ciclos de clock do próprio SoC utilizando a infraestrutura do LiteX.

Neste trabalho, um SoC foi implementado na FPGA, utilizando o `softcore` VexRiscv como processador. O LiteX gera automaticamente as bibliotecas em C necessárias para acesso às funcionalidades implementadas em hardware, incluindo registradores de controle e contadores de ciclos.

Com base nessas bibliotecas, em específico o arquivo `generated/csr.h`, foi criado um conjunto de funções (Figura 2), para interagir com os registradores do periférico `Timer` [7], disponibilizado pelo LiteX e integrado ao SoC. Essas funções são baseado em uma discussão da comunidade do projeto LiteX [8], escritas em C e encapsulando as operações de inicialização, leitura e cálculo dos ciclos consumidos entre o início e o fim de uma determinada função.

```c
#include "timer.h"

uint32_t start_ticks;
uint32_t elapsed_ticks;

uint32_t get_elapsed_ticks(void) {
    return elapsed_ticks;
}

void start_stopwatch(void) {
    // Disable timer
    timer0_en_write(0);

    // Set timer to count down from maximum value
    timer0_reload_write(0xffffffff);
    timer0_load_write(0xffffffff);

    // Enable timer
    timer0_en_write(1);

    // Update and read initial value
    timer0_update_value_write(1);
    start_ticks = timer0_value_read();
}

void stop_stopwatch(void) {
    uint32_t end_ticks;

    // Update and read final value
    timer0_update_value_write(1);
    end_ticks = timer0_value_read();

    // Calculate elapsed ticks (timer counts down)
    elapsed_ticks = start_ticks - end_ticks;
}
```

Figura 2: Funções desenvolvido para interagir com o periférico `Timer` disponibilizado pelo LiteX. Para utilizá-lo, é necessário chamar as funções `start_stopwatch` e `stop_stopwatch` em sequência, de modo a medir a quantidade de ciclos de clock. Por fim, a função `get_elapsed_ticks` retorna o número de ciclos decorrido entre essas chamadas.

Na implementação apresentada, a ideia geral é realizar uma contagem decrescente de ciclos de clock. Para isso, duas variáveis são utilizadas: `start_ticks`, que armazena a quantidade de ciclos no momento inicial da medição, e `elapsed_ticks`, que armazenará o total de ciclos transcorridos. A função `start_stopwatch` configura o temporizador para iniciar a contagem a partir do valor máximo permitido. Já a função `stop_stopwatch` atualiza e lê o valor final do contador, calculando a diferença em relação ao valor inicial. O resultado, correspondente ao número de ciclos de clock transcorridos, pode ser acessado por meio da função `get_elapsed_ticks`.

Os programas foram compilados com o compilador **gcc-riscv64-unknown-elf** versão `13.2.0`. Todos os programas foram compilados com a flag de otimização `-Os`. Essa é a flag de otimização padrão adotada pelo LiteX, como pode ser observado no arquivo `common.mak` [9], que contém as definições de compilação comuns utilizadas pelo framework.

A figura 3 mostra o código utilizado para medir o tempo necessário para executar a função `predict` cem vezes.

Os resultados da execução do código 3 podem ser observados em 4. A função utilizada para reportar o tempo de execução esta na figura 5.

```c
// First benchmark - floating point prediction (CPU)
printf("Running CPU only benchmark...\n");
start_stopwatch();

for (i = 0; i < 100; i += 1)
{
    p1 += predict(input2);
}

stop_stopwatch();
print_elapsed_time(elapsed_ticks, "CPU only Benchmark");
```

Figura 3: Código que mede o numero de ciclos necessários para executar 100 vezes a função `predict` sem acelerador.

```
Running CPU only benchmark...
=== CPU only Benchmark Results ===
Raw ticks: 400566838
Elapsed time: 00:14.835 (14835 milliseconds)
CPU: VexRiscv @ 27MHz
Clock frequency: 27000000 Hz
```

Figura 4: Resultados obtidos para a execução da função `predict` sem acelerador cem vezes.

```c
// Print elapsed time without using floats
void print_elapsed_time(uint32_t ticks, const char *benchmark_name)
{
    // Convert ticks to microseconds first, then to milliseconds
    uint32_t microseconds = ticks / (CONFIG_CLOCK_FREQUENCY / 1000000);
    uint32_t milliseconds = microseconds / 1000;
    uint32_t seconds = milliseconds / 1000;
    uint32_t minutes = seconds / 60;
    uint32_t rem_seconds = seconds % 60;
    uint32_t rem_milliseconds = milliseconds % 1000;
    uint32_t MHz = CONFIG_CLOCK_FREQUENCY / 1000000;

    printf("=== %s Results ===\n", benchmark_name);
    printf("Raw ticks: %lu\n", ticks);
    printf("Elapsed time: %02d:%02d.%03d (%lu milliseconds)\n",
           (int)minutes, (int)rem_seconds, (int)rem_milliseconds, (unsigned long)milliseconds);
    printf("CPU: %s @ %luMHz\n", CONFIG_CPU_HUMAN_NAME, (unsigned long)MHz);
    printf("Clock frequency: %lu Hz\n", (unsigned long)CONFIG_CLOCK_FREQUENCY);
    printf("\n");
}
```

Figura 5: Código exibe os resultados das medições coletadas.

6 Acelerador: Árvore de Redução Binária

O acelerador desenvolvido tem como objetivo calcular o produto escalar entre vetores de entrada e seus respectivos pesos, operação central na inferência de modelos de Regressão Logística Multinomial. A implementação foi realizada utilizando o framework LiteX, com suporte à construção de módulos em linguagem Python por meio da biblioteca `Migen` [10].

A arquitetura do acelerador é organizada em três blocos principais de registradores do tipo `CSR` (*Control and Status Registers*). O primeiro bloco, chamado `input`, armazena os valores de entrada e é composto por quatro registradores de 32 bits, totalizando 128 bits. O segundo bloco, `weight`, possui a mesma estrutura e guarda os pesos utilizados no cálculo. Por fim, o terceiro bloco contém um único registrador de 32 bits, denominado `result`, que disponibiliza o valor final do produto escalar realizado pelo hardware.

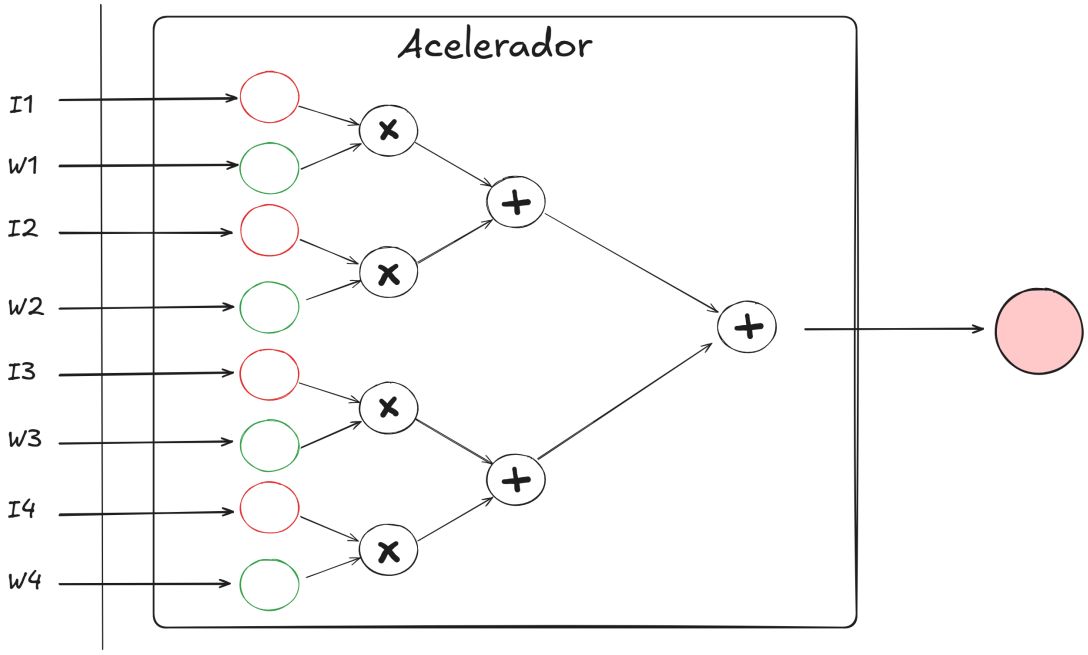

Figura 6: Diagrama em alto nível do acelerador implementado em hardware como um periféri com LiteX. A arquitetura segue o modelo de uma árvore de redução, utilizando exclusivamente lógica combinacional. Os círculos vermelhos indicam os dados de entrada, enquanto os círculos verdes representam os pesos aplicados na inferência.

O acelerador foi projetado para operar sobre os blocos de registradores da entrada e pesos, sendo o produto escalar desses dados calculado com lógica puramente combinacional. Para realizar a multiplicação elemento a elemento, pares `input_i` $\times$ `weight_i` são gerados e armazenados em sinais intermediários de 64 bits. A etapa seguinte é a soma de todos os produtos parciais. Para isso, optou-se pela implementação de uma **árvore de redução binária**, uma técnica eficiente para realizar somas em paralelo. A cada nível da árvore, pares de sinais são somados e propagados para o nível seguinte, até que reste apenas um único valor, o resultado final da operação que é truncado para 32 bits. A figura 6, representa este esquema de operação.

A abordagem adotada não é a mais eficiente em termos de área ocupada na FPGA, pois exige a utilização de múltiplos somadores, cuja quantidade cresce conforme o tamanho dos vetores de entrada. Uma alternativa mais econômica em recursos seria utilizar apenas um único somador e realizar a soma dos produtos parciais em múltiplos ciclos de clock. No entanto, a árvore de somadores é totalmente combinacional, o que permite calcular o resultado completo do produto escalar em um único ciclo, para profundidades da árvore que permitem a computação em um único ciclo. Essa escolha, embora mais custosa em termos de área, proporciona uma execução significativamente mais rápida.

Os tamanhos dos blocos de entrada e pesos foram definidos como 4 devido às limitações dos recursos da FPGA presente na plataforma Tang Nano 9K.

```python
class DotProductAccelerator(LiteXModule):
    def __init__(self, input_size=4, data_width=32):
        """
        A simple hardware accelerator for logistic regression dot product:
        output = sum(input_i * weight_i)
        """
        # CSR definitions
        self.input = CSRStorage(fields=[
            CSRField(f"value_{i}", size=data_width) for i in range(input_size)
        ])
        self.weight = CSRStorage(fields=[
            CSRField(f"weight_{i}", size=data_width) for i in range(input_size)
        ])
        self.result = CSRStatus(data_width, name="result")

        # Access fields as arrays
        input_array = Array(self.input.fields.fields)
        weight_array = Array(self.weight.fields.fields)

        # Element-wise multiply inputs and weights
        products = [
            Signal(2*data_width, name=f"product_{i}")
            for i in range(input_size)
        ]
        for i in range(input_size):
            self.comb += products[i].eq(input_array[i] * weight_array[i])

        # Build a reduction tree to sum all the products
        def pairwise_sum_level(signals, level):
            next_level = []
            for i in range(0, len(signals), 2):
                s = Signal(data_width*2, name=f"sum_l{level}_{i//2}")
                self.comb += s.eq(signals[i] + signals[i+1])
                next_level.append(s)
            return next_level

        sum_level = products
        level = 0
        while len(sum_level) > 1:
            sum_level = pairwise_sum_level(sum_level, level)
            level += 1

        self.comb += self.result.status.eq(sum_level[0][:32])

```

Figura 7: Código Python do acelerador do produto escalar implementado com LiteX.

O código da implementação do acelerador pode ser observado na figura 7. A implementação foi feita utilizando a biblioteca `Migen` em conjunto com módulos oferecidos pelo LiteX. As linhas 27 a 32 definem os produtos elemento a elemento entre pesos e dados. Os produtos são realizados assumindo que os operandos estão em uma representação de ponto fixo. Como ambos os operandos são números de 32 bits, o resultado e um sinal de 64 bits. A arvore de redução é definida nas linhas 34 a 47.

7 Resultados

A figura 8 mostra o tempo de 100 execuções da função `predict_hw` que é a função `predict` com acelerador. Comparando os resultados obtidos, o tempo medido para a execução sem acelerador foi de 14835 milissegundos, já o tempo medido com acelerador foi de 3385 milissegundos. Com base nos tempos medidos, o speedup obtido pelo acelerador pode ser estimado como sendo aproximadamente 4.38x. Esse speedup é condizente com o fato do acelerador executar simultaneamente 4 multiplicações seguido de 3 somas, ao passo que a versão sem acelerador precisar realizar todas as operações de forma sequencial.

```
Running Hardware acceleration benchmark...
=== CPU Hardware acceleration Benchmark Results ===
Raw ticks: 91397301
Elapsed time: 00:03.385 (3385 milliseconds)
CPU: VexRiscv @ 27MHz
Clock frequency: 27000000 Hz
```

Figura 8: Resultados obtidos para a execução da função `predict` com acelerador cem vezes.

8 Conclusão

Este trabalho utilizou o framework LiteX para implementar um acelerador em hardware para calcular o produto escalar de dois vetores. O LiteX oferece a capacidade de rápida prototipação e experimentação, além de diversos módulos prontos com as principais funcionalidades desejadas para um SoC. Para testar o acelerador, um modelo logístico foi treinado e portado para se executado na placa alvo. O tempo de execução sem acelerador foi medido e comparado com o tempo de execução utilizando o acelerador. Foi observado um speedup de aproximadamente 4.4x da versão com acelerador para a versão sem acelerador.

Referências

[1] LiteX Wiki. LiteX for Hardware Engineers - Installation. `https://github.com/enjoy-digital/litex/wiki/Installation`. Acesso em: 1 de junho de 2025.

[2] Lushay Labs. Lushay Labs. `https://lushaylabs.com/`. Acesso em: 1 de junho de 2025.

[3] Wikipedia. Multinomial Logistic Regression. `https://en.wikipedia.org/wiki/Multinomial_logistic_regression`. Acesso em: 29 de junho de 2025.

[4] Wikipedia. Logistic Regression. `https://en.wikipedia.org/wiki/Logistic_regression`. Acesso em: 29 de junho de 2025.

[5] MicroMLgen. MicroMLgen Repository. `https://github.com/eloquentarduino/micromlgen/tree/master`. Acesso em: 29 de junho de 2025.

[6] scikit-learn. load_digits. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html. Acesso em: 29 de junho de 2025.

[7] LiteX. Timer Peripheral Code. https://github.com/enjoy-digital/litex/blob/master/litex/soc/cores/timer.py. Acesso em: 29 de junho de 2025.

[8] James Timothy Meech. What is the most sensible way to time C code running over FemtoRV on LiteOS? https://github.com/BrunoLevy/learn-fpga/discussions/105. Acesso em: 29 de junho de 2025.

[9] LiteX. common.mak - Common Build Rules. https://github.com/enjoy-digital/litex/blob/master/litex/soc/software/common.mak. Acesso em: 29 de junho de 2025.

[10] M-Labs. Migen: A Python Toolbox for Building Complex Digital Hardware. https://m-labs.hk/migen/manual/introduction.html. Acesso em: 30 de junho de 2025.